

NOTE: This specification has not been approved and is subject to modification.  
DO NOT USE FOR ACQUISITION PURPOSES.

## **Software Communications Architecture Specification**

**MSRC-5000SCA  
V1.0  
May 17, 2000**

Prepared for the  
Joint Tactical Radio System (JTRS) Joint Program Office

Prepared by the  
Modular Software-programmable Radio Consortium  
under Contract No. DAAB15-00-3-0001

## Revision Summary

0.1	original draft for industry review and comment
0.2	updated draft for industry review and comment. Changes are extensive, involving format and content.
0.3	updated draft for industry review and comment. Changes are marked with change bars in the margins.
0.4	updated draft for industry review and comment. Significant changes: - previous sec. 3.1.3.3.6 <i>Factory</i> properly recognized as a Core Application Interface and is now 3.1.3.1.7 <i>ResourceFactory</i> . - Added material to section 3.2.2.2, Network Applications - Added material to section 5, Security Architecture Definition
1.0	Formal release for initial validation. Changes made include correction of errors and the following: - added figure and table captions; updated figure and table numbers - moved Use Cases to the SRD - deleted unused appendix references; renumbered existing appendices - changed Access to I/O where discussing signal and control to/from system - expanded and corrected examples of use of SCA in section 2 - changed Application Profile to Software Profile; included references to the Domain Profile files where used by CF interfaces - deleted "NAPI" as a unique distinction of networking APIs - removed requirements on CORBA extensions where products are not available commercially - moved brief discussion of security control related requirements from utility applications to 3.1.3 CF - Moved 3.1.3.3.6 Adapters to 2.2.1.6.2 since adapters are implementations of the CF <i>Resource</i> interface, not separate CF interfaces - moved <i>StringConsumer</i> to the Framework Service Interface section - moved 3.1.3.1.8 CF PushPort Interfaces and 3.1.3.1.9 CF PullPort Interfaces to 3.2.1.3.1 Applications – CF Interface Extensions - clarified and expanded CF interface definitions, behavior, types, etc. - added Naming context definitions - expanded definitions of Domain Profile - clarified explanations regarding use of hardware attributes - moved JTRS security requirements from section 5 to the Security Supplement to the SCA Specification - centralized discussion of registration body and UUIDs in section 7 - added appendix with Service Definition Description for APIs

Changes from the previous revision, other than editorial corrections, are marked with change bars in the margins.

## Table of Contents

<b>FOREWORD .....</b>	<b>vii</b>
<b>1 INTRODUCTION .....</b>	<b>1-1</b>
1.1 Scope. ....	1-1
1.2 Compliance.....	1-2
1.2.1 Joint Technical Architecture Compliance. ....	1-2
1.3 Document conventions, Terminology, and Definitions. ....	1-2
1.3.1 Conventions and Terminology. ....	1-2
1.3.1.1 Unified Modeling Language.....	1-2
1.3.1.2 Interface Definition Language.....	1-2
1.3.1.3 eXtensible Markup Language.....	1-3
1.3.1.4 Color Coding. ....	1-3
1.3.1.5 Requirements Language. ....	1-3
1.3.1.6 CF Interface and Operation Identification.....	1-3
1.3.2 Definitions. ....	1-3
1.4 Document Content.....	1-3
<b>2 OVERVIEW.....</b>	<b>2-1</b>
2.1 Architecture Definition Methodology.....	2-1
2.2 Architecture Overview.....	2-1
2.2.1 Overview - Software Architecture.....	2-1
2.2.1.1 Bus Layer (Board Support Package). ....	2-2
2.2.1.2 Network & Serial Interface Services. ....	2-2
2.2.1.3 Operating System Layer. ....	2-2
2.2.1.4 Core Framework. ....	2-3
2.2.1.5 CORBA Middleware.....	2-3
2.2.1.6 Application Layer. ....	2-3
2.2.1.6.1 Applications. ....	2-4
2.2.1.6.2 Adapters. ....	2-4
2.2.1.7 Software Radio Functional Concepts. ....	2-5
2.2.1.7.1 Software Reference Model.....	2-5
2.2.1.7.2 <i>ModemDevice</i> Functionality. ....	2-7
2.2.1.7.3 <i>NetworkResource</i> and <i>LinkResource</i> Functionality. ....	2-8
2.2.1.7.4 <i>I/ODevice</i> Functionality. ....	2-9
2.2.1.7.5 <i>SecurityDevice</i> Functionality. ....	2-10
2.2.1.7.6 <i>UtilityResource</i> Functionality. ....	2-11
2.2.1.8 System Control. ....	2-11
2.2.2 Networking Overview. ....	2-12
2.2.2.1 External Networking Protocols. ....	2-13
2.2.2.2 SCA Support for External Networking Protocols. ....	2-14
2.2.3 Overview - Hardware Architecture. ....	2-15

<b>3</b>	<b>SOFTWARE ARCHITECTURE DEFINITION .....</b>	<b>3-1</b>
<b>3.1</b>	<b>Operating Environment.....</b>	<b>3-1</b>
3.1.1	Operating System.....	3-1
3.1.2	Middleware & Services.....	3-2
3.1.2.1	CORBA.....	3-2
3.1.2.2	CORBA Extensions.....	3-2
3.1.2.2.1	Naming Service.....	3-2
3.1.2.2.2	Quality of Service Control.....	3-2
3.1.3	Core Framework.....	3-2
3.1.3.1	Base Application Interfaces.....	3-4
3.1.3.1.1	<i>Port</i> .....	3-4
3.1.3.1.2	<i>LifeCycle</i> .....	3-6
3.1.3.1.3	<i>TestableObject</i> .....	3-7
3.1.3.1.4	<i>PropertySet</i> .....	3-8
3.1.3.1.5	<i>Resource</i> .....	3-10
3.1.3.1.6	<i>ResourceFactory</i> .....	3-13
3.1.3.2	Framework Control Interfaces.....	3-16
3.1.3.2.1	<i>Application</i> .....	3-16
3.1.3.2.2	<i>ApplicationFactory</i> .....	3-19
3.1.3.2.3	<i>DomainManager</i> .....	3-24
3.1.3.2.4	<i>Device</i> .....	3-32
3.1.3.2.5	<i>DeviceManager</i> .....	3-40
3.1.3.3	Framework Services Interfaces.....	3-44
3.1.3.3.1	<i>File</i> .....	3-44
3.1.3.3.2	<i>FileSystem</i> .....	3-47
3.1.3.3.3	<i>FileManager</i> .....	3-52
3.1.3.3.4	<i>StringConsumer</i> .....	3-56
3.1.3.3.5	<i>Logger</i> .....	3-57
3.1.3.3.6	<i>Timer</i> .....	3-66
3.1.3.4	Domain Profile.....	3-66
3.1.3.4.1	Software Package Descriptor.....	3-67
3.1.3.4.2	Software Component Descriptor.....	3-67
3.1.3.4.3	Software Assembly Descriptor.....	3-68
3.1.3.4.4	Property File.....	3-68
3.1.3.4.5	Device Package Descriptor File.....	3-68
3.1.3.4.6	Device Assembly Descriptor.....	3-68
<b>3.2</b>	<b>Applications.....</b>	<b>3-69</b>
3.2.1	General Application Requirements.....	3-69
3.2.1.1	OS Services.....	3-69
3.2.1.2	CORBA Services.....	3-69
3.2.1.3	CF Interfaces.....	3-69
3.2.1.3.1	CF Interface Extensions.....	3-69
3.2.2	Application Interfaces.....	3-71
3.2.2.1	Utility Applications.....	3-71
3.2.2.1.1	Installer Utility.....	3-71
3.2.2.2	Service APIs.....	3-73

3.2.2.2.1	Service Definitions.....	3-73
3.2.2.2.2	API Transfer Mechanisms.....	3-75
<b>3.3</b>	<b>General Software Rules.....</b>	<b>3-78</b>
3.3.1	Software Development Languages.....	3-78
3.3.1.1	New Software. ....	3-78
3.3.1.2	Legacy Software. ....	3-78
<b>4</b>	<b>HARDWARE ARCHITECTURE DEFINITION.....</b>	<b>4-1</b>
<b>4.1</b>	<b>Basic Approach.....</b>	<b>4-1</b>
<b>4.2</b>	<b>Class Structure.....</b>	<b>4-1</b>
4.2.1	Top Level Class Structure. ....	4-2
4.2.2	<i>HWModule(s)</i> Class Structure. ....	4-3
4.2.3	Class Structure with Extensions. ....	4-4
4.2.3.1	<i>RF</i> Class Extension.....	4-4
4.2.3.2	<i>Modem</i> Class Extension.....	4-6
4.2.3.3	<i>Processor</i> Class Extension. ....	4-7
4.2.3.4	<i>INFOSEC</i> Class. ....	4-8
4.2.3.5	<i>I/O</i> Class Extension. ....	4-9
4.2.4	Attribute Composition. ....	4-10
<b>4.3</b>	<b>Domain Criteria.....</b>	<b>4-10</b>
<b>4.4</b>	<b>Performance Related Issues.....</b>	<b>4-10</b>
<b>4.5</b>	<b>General Hardware Rules. ....</b>	<b>4-11</b>
4.5.1	Device Profile. ....	4-11
4.5.2	Hardware Critical Interfaces.....	4-11
4.5.2.1	Interface Definition.....	4-11
4.5.2.2	Interface Standards. ....	4-11
4.5.2.2.1	Interface Selection.....	4-11
4.5.3	Form Factor. ....	4-11
4.5.4	Modularity. ....	4-11
<b>5</b>	<b>SECURITY ARCHITECTURE DEFINITION.....</b>	<b>5-1</b>
<b>6</b>	<b>COMMON SERVICES AND DEPLOYMENT CONSIDERATIONS.....</b>	<b>6-1</b>
<b>6.1</b>	<b>Common System Services. ....</b>	<b>6-1</b>
<b>6.2</b>	<b>Operational and Deployment Considerations.....</b>	<b>6-1</b>
<b>7</b>	<b>ARCHITECTURE COMPLIANCE.....</b>	<b>7-1</b>
<b>7.1</b>	<b>Certification Authority.....</b>	<b>7-1</b>
<b>7.2</b>	<b>Responsibility for Compliance Evaluation.....</b>	<b>7-1</b>
<b>7.3</b>	<b>Evaluating Compliance. ....</b>	<b>7-1</b>
<b>7.4</b>	<b>Registration. ....</b>	<b>7-1</b>

**APPENDIX A. GLOSSARY**

**APPENDIX B. SCA APPLICATION ENVIRONMENT PROFILE**

**APPENDIX C. CORE FRAMEWORK IDL**

**APPENDIX D. DOMAIN PROFILE**

**APPENDIX E. SERVICE DEFINITION DESCRIPTION**

## List of Figures

Figure 1-1. The Architecture Framework and its Relationship to Implementation.....	1-2
Figure 1-2. Color Coding Used in Document Figures.....	1-3
Figure 2-1. Software Structure .....	2-2
Figure 2-2. Example Message Flows with and without Adapters.....	2-5
Figure 2-3. Software Reference Model .....	2-5
Figure 2-4. Conceptual Model of Resources .....	2-6
Figure 2-5. Example of Modem Resources .....	2-7
Figure 2-6. Example of Networking Resources .....	2-8
Figure 2-7. Examples of I/O Resources.....	2-9
Figure 2-8. Examples of Security Devices and Resources .....	2-10
Figure 2-9. Example of Utility Resources .....	2-11
Figure 2-10. External Network Protocols and SCA Support.....	2-12
Figure 2-11. SCA-Supported Networking Mapped to OSI Network Model.....	2-14
Figure 2-12. Hardware Architecture Framework .....	2-16
Figure 3-1. Notional Relationship of OE and Application to the SCA AEP.....	3-1
Figure 3-2. Core Framework IDL Relationships.....	3-3
Figure 3-3. <i>Port</i> Interface UML .....	3-4
Figure 3-4. <i>LifeCycle</i> Interface UML .....	3-6
Figure 3-5. <i>TestableObject</i> Interface UML .....	3-7
Figure 3-6. <i>PropertySet</i> Interface UML .....	3-9
Figure 3-7. <i>Resource</i> Interface UML.....	3-11
Figure 3-8. <i>ResourceFactory</i> Interface UML.....	3-14
Figure 3-9. <i>Application</i> Interface UML.....	3-17
Figure 3-10. <i>Application</i> Behavior .....	3-19
Figure 3-11. <i>ApplicationFactory</i> UML .....	3-20
Figure 3-12. <i>ApplicationFactory</i> Behavior.....	3-24
Figure 3-13. <i>DomainManager</i> Interface UML.....	3-25
Figure 3-14. <i>DomainManager</i> Sequence Diagram for <i>RegisterDeviceManager</i> Operation.....	3-27
Figure 3-15. <i>DomainManager</i> Sequence Diagram for <i>RegisterDevice</i> Operation.....	3-29
Figure 3-16. <i>Device</i> Interface UML .....	3-33
Figure 3-17. <i>DeviceManager</i> UML.....	3-41
Figure 3-18. <i>File</i> Interface UML.....	3-44
Figure 3-19. <i>FileSystem</i> Interface UML.....	3-47
Figure 3-20. <i>FileManager</i> Interface UML .....	3-53
Figure 3-21. <i>StringConsumer</i> Interface UML .....	3-56
Figure 3-22. <i>Logger</i> Interface UML.....	3-57
Figure 3-23. LogData Operational Behavior.....	3-61
Figure 3-24. Relationship of Domain Profile XML File Types .....	3-67
Figure 3-25. PushPort Data Interfaces.....	3-70
Figure 3-26. PullPort Data Interfaces .....	3-71
Figure 3-27. Device Installation Sequence Diagram.....	3-72
Figure 3-28. Software Installation Sequence Diagram.....	3-73
Figure 3-29. Reusing an Existing Service Definition Without an IDL Interface .....	3-75
Figure 3-30. Standard and Alternate Transfer Mechanism .....	3-76
Figure 4-1. Top Level Hardware Class Structure.....	4-2

Figure 4-2. Hardware Module Class Structure ..... 4-3

Figure 4-3. RF Class Extension..... 4-5

Figure 4-4. Modem Class Extension ..... 4-6

Figure 4-5. Processor Class ..... 4-7

Figure 4-6. INFOSEC Class ..... 4-8

Figure 4-7. I/O Class Extension ..... 4-9

Figure 4-8. Typcial Hardware Device Description using the SCA HW Class Structure ..... 4-10

**List of Tables**

Table 3-1. *Logger*, Consumers, and Producers Log Levels ..... 3-59



## Foreword

**Introduction.** The Software Communication Architecture (SCA) specification is being published by the Joint Tactical Radio System (JTRS) Joint Program Office (JPO). This program office was established to pursue the development of future communication systems, capturing the benefits of the technology advances of recent years which are expected to greatly enhance interoperability of communication systems and reduce development and deployment costs. The goals set for the program are:

- Greatly increased operational flexibility and interoperability of globally deployed systems
- Reduced supportability costs
- Upgradeability in terms of easy technology insertion and capability upgrades
- Reduced system acquisition and operation cost

In order to achieve these goals, the SCA has been structured to

- provide for portability of applications software between different SCA implementations
- leverage commercial standards to reduce development cost
- reduce development time of new waveforms through the ability to reuse design modules
- build on evolving commercial frameworks and architectures

The SCA is deliberately designed to meet commercial application requirements as well as more stringent military applications. It is the expectation of the Government that the SCA will become a commercially approved standard. It is for this reason that a wide cross section of industry has been invited to participate in the development and the validation of the SCA. The SCA is not a system specification, as it is intended to be implementation independent, but a set of rules that constrain the design of systems to achieve the objectives listed above. The SCA specification version 1.0 establishes the baseline for architecture validation and for future development. The validation of the SCA includes the demonstration that multiple vendors can independently design systems, which, when built according to the SCA requirements, meet the program goals outlined above. Lessons learned during the validation will be incorporated into future SCA releases. SCA version 2.0 is planned for release in November 2000.

The SCA documentation consists of the basic architecture specification containing all requirements necessary for general, commercial implementation, and supplements which contain military-unique requirements pertaining to security and application implementation.

**Software Structure.** The software framework of the SCA defines the Operating Environment (OE) and specifies the services and interfaces that applications use from that environment. The OE is made of:

- a Core Framework (CF),
- a software transfer mechanism called CORBA, and
- an Operating System (OS) with associated board support packages.

The OE imposes design constraints on waveform and other applications to provide increased portability of those applications from one SCA-compliant radio platform to another. These design constraints include specified interfaces between the Core Framework and application software, and restrictions on waveform usage of the Operating System. This approach also provides a building-block structure for defining application programming interfaces (APIs) between application software components. This building-block structure for API definition facilitates component-level reuse and allows significant flexibility for developers to define waveform-specific APIs.

The SCA makes use of object-oriented (OO) design to define the software structure and represents the interface and service definitions in terms of classes and inheritance of the OO approach. The SCA does not impose any one specific structure on software applications, although many of the examples cited do partition applications into radio associated elements like Modem, Link, Network, Security, and Host Interface. These illustrate the concepts of inheritance from base classes, interfaces and services of the CF.

**Hardware Structure.** The hardware framework also uses OO concepts to define typical partitions of real systems. The primary purpose of the hardware structure is to require complete and comprehensive publication of interfaces and attributes once systems have been built. With these published specifications, additional vendors can provide modules within a system and software developers can identify hardware modules with capabilities required for a particular waveform application. Hardware modularity also facilitates technology insertion as future programmable elements increase in capability.

**Military Applications.** To maximize the commercial application of the SCA and benefit from advances that will accrue, military-unique system requirements are in supplemental SCA documentation. These Supplements to the SCA Specification include:

- security requirements to insure adequate protection of military secure communications and facility certifications of JTRS product systems from the NSA, and
- an Application Program Interface (API) structure associated with radio system services such as modem, networking, security, and external interfaces. These APIs, when fully defined, improve portability of applications within JTRS implementations, and make reuse of functional components of those applications easier. For example, standardizing APIs for a security module within a JTRS enables reuse of common modules for multiple waveform applications. Standardizing networking APIs improves portability of networking applications and offers easier internetworking functions such as routing, bridging and providing gateways. *{This Supplement is being developed and is not currently available.}*

An additional accompanying document, the Support and Rationale Document (SRD), provides the rationale behind architectural decisions along with further supporting material. The SRD is planned for initial release in June 2000.

**Future Directions.** The next major release of the SCA, v2.0, is planned at the end of the ongoing prototyping and validation phase. This release will incorporate lessons learned from prototyping and validation, as well as input received from industry review of v1.0.

The JTRS JPO intends to extend the SCA to further define application structure in the form of required functional partitions for military waveform software applications and hardware modules. This will take the form of APIs noted above and improve portability and interoperability for military applications.

**Feedback.** An open architecture framework is greatly improved through active feedback and recommended changes from a wide audience of potential users. The JTRS JPO solicits and encourages feedback to this document and provides a form available from <http://www.jtrs.sarda.army.mil/docs/documents/sca.html>. Send the completed form to [jtrs.sca@sarda.army.mil](mailto:jtrs.sca@sarda.army.mil). Recommended additions to the SCA must be unencumbered by copyright restrictions or intellectual property rights. Changes to the SCA are controlled by a jointly-chaired JTRS JPO and industry Configuration Control Board (CCB).



# 1 INTRODUCTION

The Software Communications Architecture (SCA) specification establishes an implementation-independent framework with baseline requirements for the development of Joint Tactical Radio System (JTRS) software configurable radios. These requirements are comprised of interface specifications, application program interfaces (APIs), behavioral specifications, and rules. The goal of this specification is to ensure the portability and configurability of the software and hardware and to ensure interoperability of products developed using the SCA.

Companion documents to this specification are Supplements to the SCA and the SCA Support and Rationale Document (SRD). The Supplements provide specific service and application interface requirements (for Security, networking, other services). The SRD provides the rationale for the SCA and examples to illustrate the implementation of the architecture for differing domains/platforms and selected waveforms.

## 1.1 SCOPE.

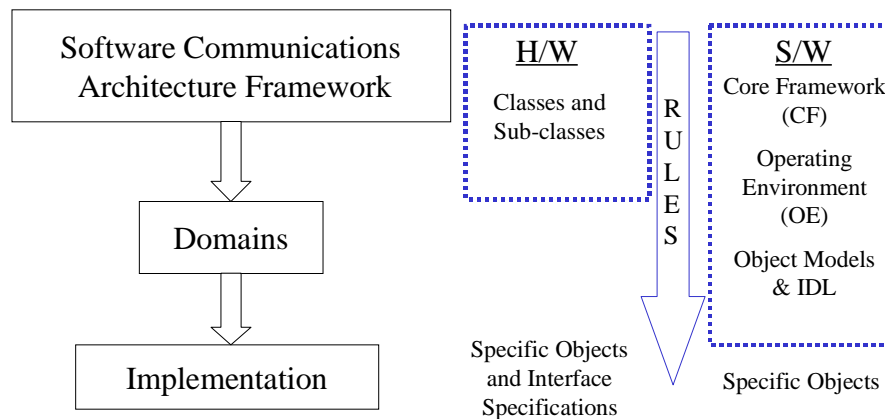
This document provides a complete definition of the SCA. It is an Architecture Framework in that it is precise in areas where reusability is effected and it is general in other areas so that unique requirements of implementations determine the specific application of the architecture. The SCA defines the hardware and software at different levels of detail to allow the broadest reusability and portability of components.

For hardware, the physical and environmental differences across domains are so diverse that physical commonality cannot be achieved for all implementations. However, by using an Object-Oriented (OO) description for the hardware, represented as hardware classes, all potential system implementations are included within a single framework. That framework has attributes (i.e., behavior and interfaces) that are applicable across those different implementations.

The architecture for software makes extensive use of object modeling and is contained in the definition of a Core Framework (CF), an integral part of a system's Operating Environment (OE). Constraints on the software development, imposed by the architecture, are on the interfaces and the structure of the software and not on the implementation of the functions that are performed. In this way, innovative designs can be put forward with appropriate protection of the developer's intellectual property and still reap the benefits of wide reuse in other implementations of the architecture. The SCA permits either hardware or software to be used in implementing a required function. The approach taken also permits legacy solutions to be incorporated, where appropriate, by encapsulation techniques to provide a "one-sided" standard interface into architecture interfaces.

This architecture specifies rules that further constrain implementations to adhere to open system standards. Specific implementation requirements may augment the rule-set to increase reusability within and across domains.

Figure 1-1 illustrates the concept of the SCA and its implementation down to specific platforms. The hardware definition stays at a framework level with rules providing implementation guidance down into domains and platforms. The software definition can be applied directly down to implementation because of its general independence from hardware implementation. There are special cases where size, weight, and power requirements limit the direct application of software objects. However, even in these cases, reusability of designs, captured in software and firmware modeling and simulation tools, reduces the cost of implementation and the development time.



**Figure 1-1. The Architecture Framework and its Relationship to Implementation**

## 1.2 COMPLIANCE.

The interfaces, behavior, and rules that define compliance with the SCA are identified in, and are an integral part of this specification. These elements are selected to maximize portability, interoperability, and configurability of the software and hardware while allowing a procurer the flexibility to address domain requirements and restrictions. If any requirements stated in this specification are in conflict with existing standards/specifications, this specification takes precedence.

### 1.2.1 Joint Technical Architecture Compliance.

The Joint Technical Architecture (JTA) mandates the minimum set of standards and guidelines for all DoD Command, Control, Communications, Computers, and Intelligence (C<sup>4</sup>I) systems acquisition. A foremost objective of the JTA is to improve and facilitate the ability of systems to support joint and combined operations in an overall investment strategy. The SCA Operating Environment fully complies with the JTA and provides a JTA-compliant framework for waveforms and other applications.

## 1.3 DOCUMENT CONVENTIONS, TERMINOLOGY, AND DEFINITIONS.

### 1.3.1 Conventions and Terminology.

#### 1.3.1.1 Unified Modeling Language.

The Unified Modeling Language (UML), defined by the Object Management Group (OMG), is used to graphically represent SCA interfaces, scenarios, use cases, and collaboration diagrams.

#### 1.3.1.2 Interface Definition Language.









Interface Definition Language (IDL), also defined by the OMG, is used to define the SCA interfaces. IDL is programming language independent and can be compiled into programming languages such as C++, Ada, and Java.

#### 1.3.1.3 eXtensible Markup Language.

eXtensible Markup Language (XML) is used in a Domain Profile to identify the capabilities, properties, inter-dependencies, and location of the hardware devices and software components that make up an SCA-compliant system

#### 1.3.1.4 Color Coding.

Color-coding is used to differentiate between architecture elements and applications in diagrams as shown in figure 1-2.

	Core Framework (CF) elements
	Commercial-Off-The-Shelf (COTS) components
	Host Applications
	Red Side Network and Link Applications
	Security Applications
	Black Side Network and Link Applications
	Modem Applications
	RF

**Figure 1-2. Color Coding Used in Document Figures**

#### 1.3.1.5 Requirements Language.

Interfaces, behavior, and rules that are imposed by this specification appear in sections 3 through 5 and are indicated by the word "shall". Editorial notes are contained within brackets and are italicized (*{example}*).

#### 1.3.1.6 CF Interface and Operation Identification.

CF interfaces and their operations are presented in italicized text.

#### 1.3.2 Definitions.

Definitions are included in Appendix A.

### **1.4 DOCUMENT CONTENT.**

This document provides an overview of the SCA in section 2, followed by the Software, Hardware, and Security architecture requirements in sections 3 – 5. Section 6 addresses requirements not contained in those functional categories. Evaluation criteria for product compliance to this specification are addressed in section 7.

Appendices include a glossary, a complete listing of CF IDL, and details of architecture requirements introduced in the main document.



## 2 OVERVIEW

This Section presents an overview of the SCA. Emphasis is on identifying the components of the architecture and the manner in which these components interact. Technical details and requirements of the architecture are contained in Sections 3 - 5.

### 2.1 ARCHITECTURE DEFINITION METHODOLOGY.

The architecture has been developed using an object-oriented approach wherein the process can be continued beyond the framework definition to product development. UML is used to graphically represent interfaces while IDL is used to define them; both have been generated using standard software development tools, allowing product development to continue directly from the architecture definition.

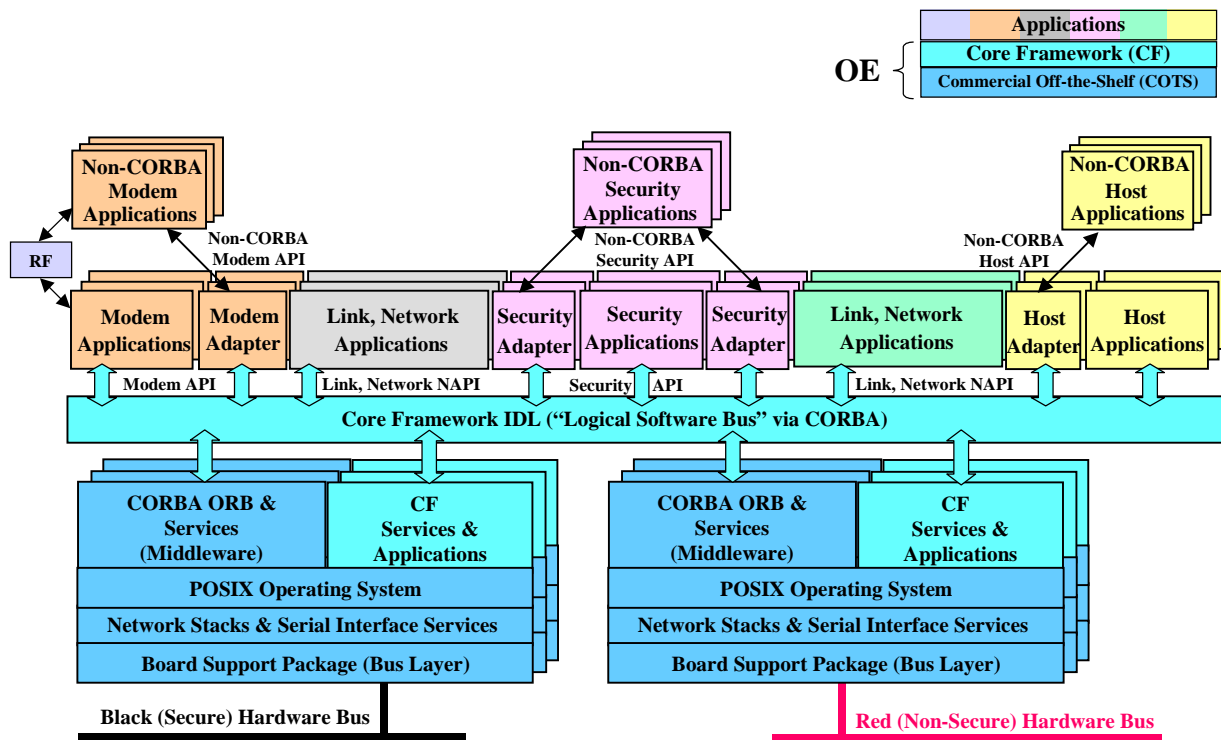
### 2.2 ARCHITECTURE OVERVIEW.

#### 2.2.1 Overview - Software Architecture.

The structure of the software architecture is shown in figure 2-1. The key benefits of the software architecture are that it:

1. Maximizes the use of commercial protocols and products,
2. Isolates both core and non-core applications from the underlying hardware through multiple layers of open, commercial software infrastructure, and
3. Provides for a distributed processing environment through the use of the Common Object Request Broker Architecture (CORBA) to provide software application portability, reusability, and scalability.

The software architecture defines an Operating Environment (OE) with the combined set of CF services and infrastructure software (including board support packages, operating system and services, and CORBA Middleware services) integrated in an SCA implementation. The software partitions that illustrate applications are typical of how waveforms might be implemented using the SCA.



**Figure 2-1. Software Structure**

#### 2.2.1.1 Bus Layer (Board Support Package).

The Software Architecture is capable of operating on commercial bus architectures. The OE supports reliable transport mechanisms, which may include error checking and correction at the bus support level. Possible buses include VME, PCI, CompactPCI, Firewire (IEEE-1394), and Ethernet. The OE does not preclude the use of different bus architectures on the Red and Black subsystems.

#### 2.2.1.2 Network & Serial Interface Services.

The Software Architecture relies on commercial components to support multiple unique serial and network interfaces. Possible serial and network physical interfaces include RS-232, RS-422, RS-423, RS-485, Ethernet, and 802.x. To support these interfaces, various low-level network protocols may be used. They include PPP, SLIP, LAPx, and others. Elements of waveform networking functionality may also exist at the Operating System layer. An example of this would be a commercial IP stack that performs routing between waveforms.

#### 2.2.1.3 Operating System Layer.

The Software Architecture includes real-time embedded operating system functions to provide multi-threaded support for applications (including CF applications). The architecture requires a standard operating system interface for operating system services in order to facilitate portability of applications.

Portable Operating System Interface (POSIX<sup>®</sup>) is an accepted industry standard. POSIX and its real-time extensions are compatible with the requirements to support the OMG CORBA specification. Complete POSIX compliance encompasses more features than are necessary to control a typical implementation. Therefore, this specification defines a minimal POSIX profile to meet SCA requirements. The SCA POSIX profile is based upon the Real-time Controller System Profile (PSE52) as defined in POSIX 1003.13.

#### 2.2.1.4 Core Framework.

The CF is the essential (“core”) set of open application-layer interfaces and services to provide an abstraction of the underlying software and hardware layers for software application designers. Section 3 presents the complete definition of all services and interfaces of the CF. The CF consists of:

1. Base Application Interfaces (*Port*, *LifeCycle*, *TestableObject*, *PropertySet*, *ResourceFactory*, and *Resource*) that can be used by all software applications,
2. Framework Control Interfaces (*Application*, *ApplicationFactory*, *DomainManager*, *Device*, and *DeviceManager*) that provide control of the system,
3. Framework Services Interfaces that support both core and non-core applications (*File*, *FileSystem*, *FileManager*, *StringConsumer*, *Logger*, and *Timer*), and
4. A Domain Profile that describes the properties of hardware devices (Device Profile) and software components (Software Profile) in the system.

The Domain Profile supports the combination of resources to create applications. Device Profile and Software Profile files utilize an XML vocabulary to describe specific characteristics of either software or device components with regard to their interfaces, functional capabilities, logical location, inter-dependencies, and other pertinent parameters.

#### 2.2.1.5 CORBA Middleware.

CORBA is used in the CF as the message passing technique for the distributed processing environment. CORBA is a cross-platform framework that can be used to standardize client/server operations when using distributed processing. Distributed processing is a fundamental aspect of the system architecture and CORBA is a widely used “Middleware” service for providing distributed processing.

All CF interfaces are defined in IDL. The CORBA protocol provides message marshalling to handle the bit packing and handshaking required for delivering the message. The SCA IDL defines operations and attributes that serve as a contract between components.

#### 2.2.1.6 Application Layer.

Applications perform user communication functions that include modem-level digital signal processing, link-level protocol processing, network-level protocol processing, internetwork routing, external input/output (I/O) access, security, and embedded utilities. Applications are required to use the CF interfaces and services. Applications' direct access to the Operating System (OS) is limited to the services specified in the SCA POSIX Profile. Networking functionality that may be implemented below the application layer, such as a commercial IP network layer, is not limited to the SCA POSIX Profile since it exists in the OS kernel space.

---

<sup>®</sup> POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

#### 2.2.1.6.1 Applications.

Applications consist of one or more *Resources*. The *Resource* interface provides a common API for the control and configuration of a software component. The application developers can extend these definitions by creating specialized *Resource* interfaces for the application. At a minimum, the extension inherits the *Resource* interface. Examples of *Resource* extensions are: *LinkResource*, *NetworkResource*, and *UtilityResource*.

*Devices* are types of *Resources* used by applications as software proxies for actual hardware devices. *ModemDevice*, *I/ODevice*, and *SecurityDevice* are examples that implement the *Device* interfaces.

*ModemDevice*, *LinkResource*, *SecurityDevice*, *I/ODevice*, and *NetworkResource* are base application interface extensions that implement APIs for waveform and networking applications.

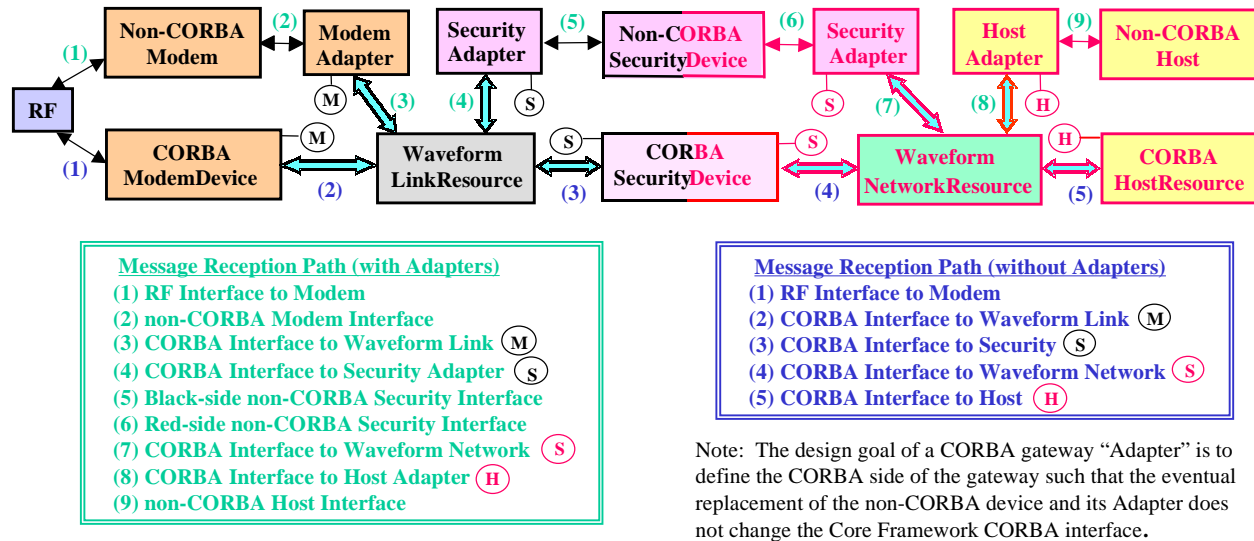
The design of a *Resource*'s internal functionality is not dictated by the Software Architecture. This is left to the application developer. Core applications, which are a part of the CF, support the non-core applications by providing the necessary function of control as well as standard interface definitions. The interfaces by which a *Resource* is controlled and communicates with other *Resources* are defined in section 3.

#### 2.2.1.6.2 Adapters.

Adapters are *Resources* or *Devices* used to support non-CORBA-capable elements. Adapters are used in an implementation to provide the translation between non-CORBA-capable components or devices and CORBA-capable *Resources*. The Adapter concept is based on the industry-accepted Adapter design pattern<sup>1</sup>. Since an Adapter implements the CF CORBA interfaces known to other CORBA-capable *Resources*, the translation service is transparent to the CORBA-capable *Resources*. Adapters become particularly useful to support non-CORBA-capable Modem, Security, and Host processing elements. Figure 2-2 depicts an example of message reception flow through the system with and without the use of Adapters. Modem, Security, and Host Adapters implement the interfaces marked by the circled letters M, S, and H respectively. Notice that the Waveform Link and Network *Resources* are unaffected by the inclusion or exclusion of the Adapters. The interface to these *Resources* remains the same in either case.

---

<sup>1</sup> “Design Patterns : Elements of Reusable Object-Oriented Software” (Addison-Wesley Professional Computing) Gamma, Helm, Johnson, and Vlissides, pg. 139



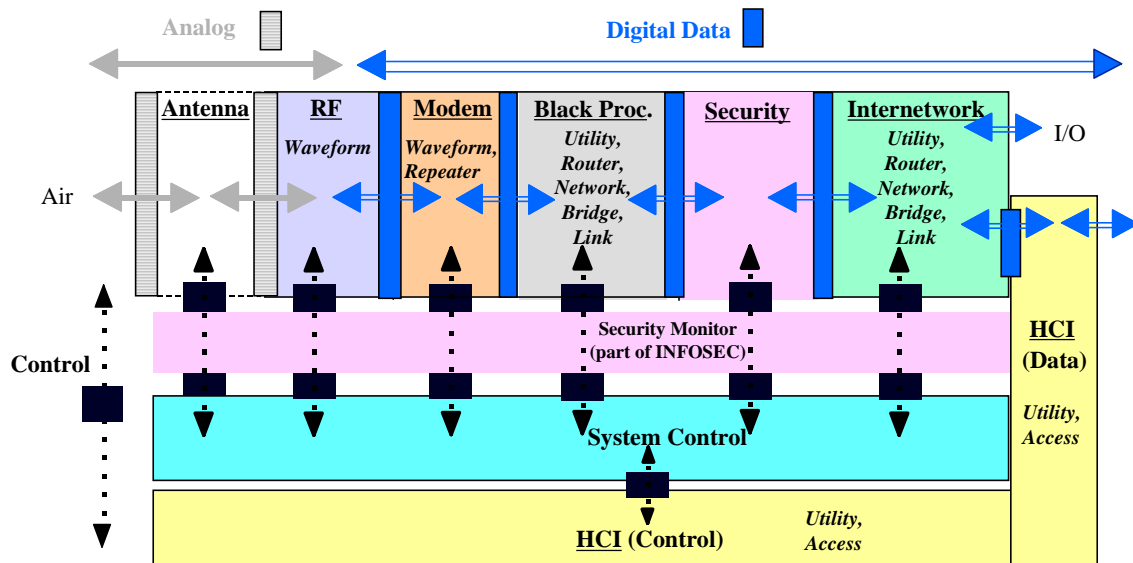
**Figure 2-2. Example Message Flows with and without Adapters**

### 2.2.1.7 Software Radio Functional Concepts.

#### 2.2.1.7.1 Software Reference Model.

The software reference model depicted in figure 2-3 is based upon the Programmable Modular Communication System (PMCS) Reference Model. This model forms a basis for the SCA by:

1. Introducing the various functional roles performed by software entities without dictating a structural model of these elements, and
2. Introducing the control and traffic data interfaces between the functional software entities.



**Figure 2-3. Software Reference Model**

The Reference Model identifies relevant functionality but does not dictate the architecture. The SCA realizes the Software Reference Model by defining a standard unit of functionality called a *Resource*. All applications are comprised of *Resources* and using *Devices*. Specific resources and devices can be identified corresponding to the functional entities of the Software Reference Model:

*ModemDevice*: addresses Antenna, RF, and Modem entities,  
*LinkResource*: addresses Black Processing entity,  
*SecurityDevice*: addresses Security entity,  
*NetworkResource*: addresses Internetworking entity,  
*I/ODevice*: addresses external interfaces such as serial, ethernet, and audio  
*UtilityResource*: addresses non-Waveform functionality.

System control entity functionality is addressed by the core framework applications: *Application*, *ApplicationFactory*, *DomainManager*, *Device*, and *DeviceManager*. Control functionality may also be localized in individual resources.

As shown in figure 2-4, all *Resources* and *Devices* inherit three Base Application Interfaces. The operations and attributes provided by *LifeCycle*, *TestableObject*, and *PropertySet* establish a common approach for interacting with any resource in a SCA environment. *Port* can be used for pushing or pulling messages between *Resources* and *Devices*. A *Resource* may consist of zero or more input and output message ports. The figure also shows examples of more specialized resources and devices that result in specific functionality for each of six example types. Clarification of the functionality associated with each of those is provided in the following subsections.

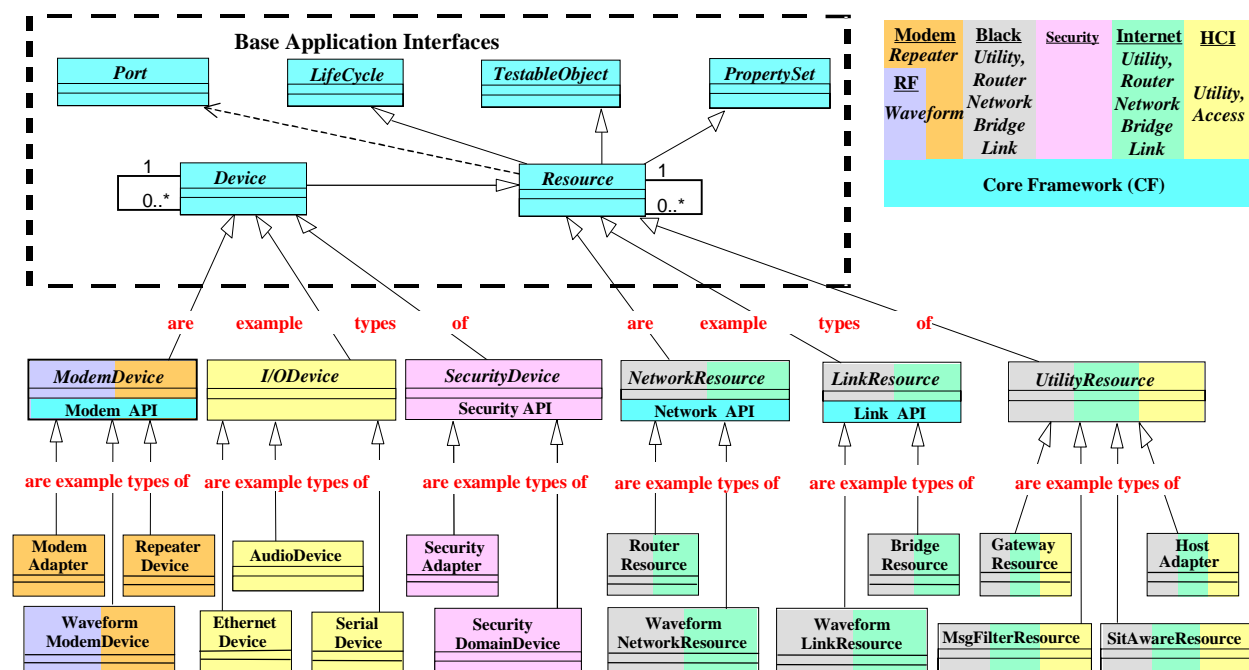


Figure 2-4. Conceptual Model of Resources

### 2.2.1.7.2 *ModemDevice* Functionality.

The *ModemDevice* provides a standard for the control and interface of a modem, which encapsulates diverse implementations of smart antenna, RF, and modem functions. The base application interfaces are extended to modem devices through a Modem API, which provides a standard interface for control and communication with modem operations from a higher (link layer) resource. The functions, performed by the *ModemDevices*, will vary depending on waveform requirements as well as hardware/software allocation and are not dictated by the CF. Typical RF and modem functions are depicted in figure 2-5.

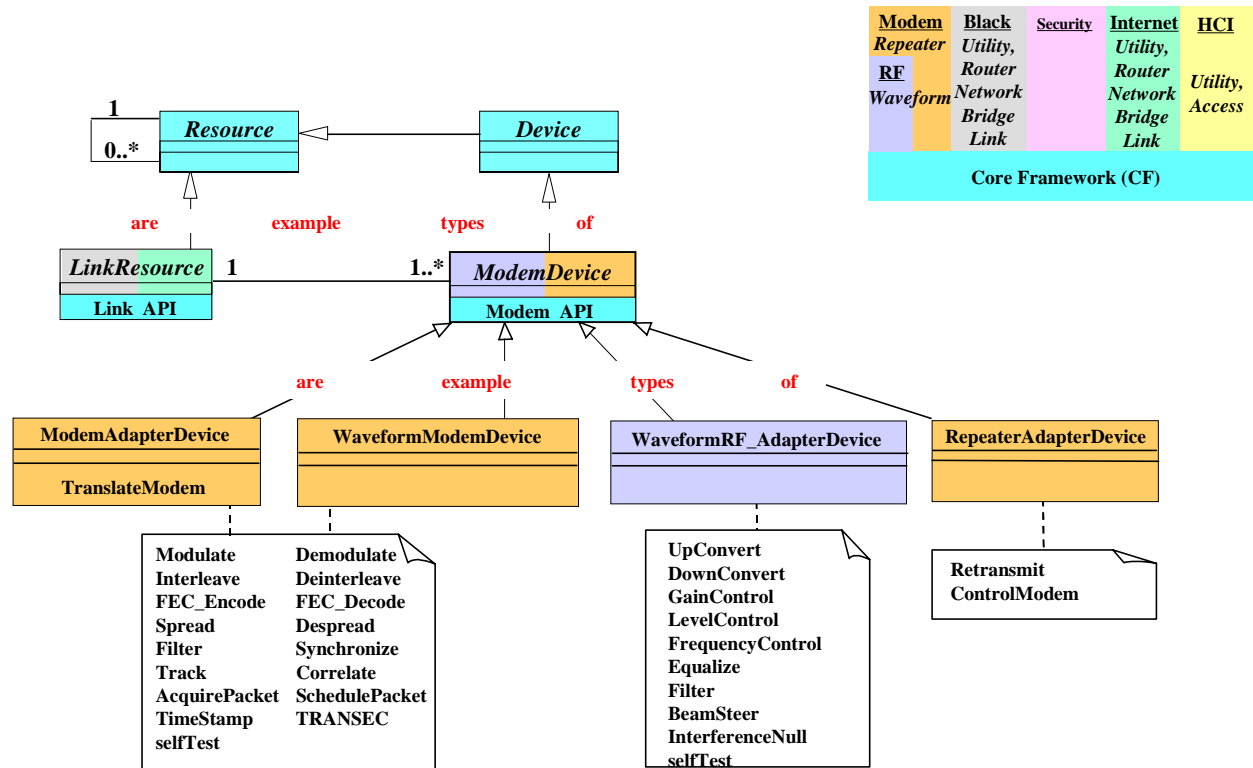


Figure 2-5. Example of Modem Resources

### 2.2.1.7.3 *NetworkResource* and *LinkResource* Functionality.

An example of networking resources is shown in figure 2-6. The CF base application interfaces are extended to link layer and network layer resources through Networking APIs (see section 2.2.2.2), provided to enable information transfer and support of specific service characteristics for networking applications. Examples are the Link API and Network API, which provide standard interfaces for control and communication between link, network, and transport layer resources.

The functions performed by the waveform networking and internetworking resources (examples shown in note boxes in figure 2-6) will vary depending on waveform requirements as well as networking requirements and are not dictated by the CF. *Resources* that provide networking behavior, including repeater, link, bridge, network, router, and gateway operations, are representative and not defined in the SCA.

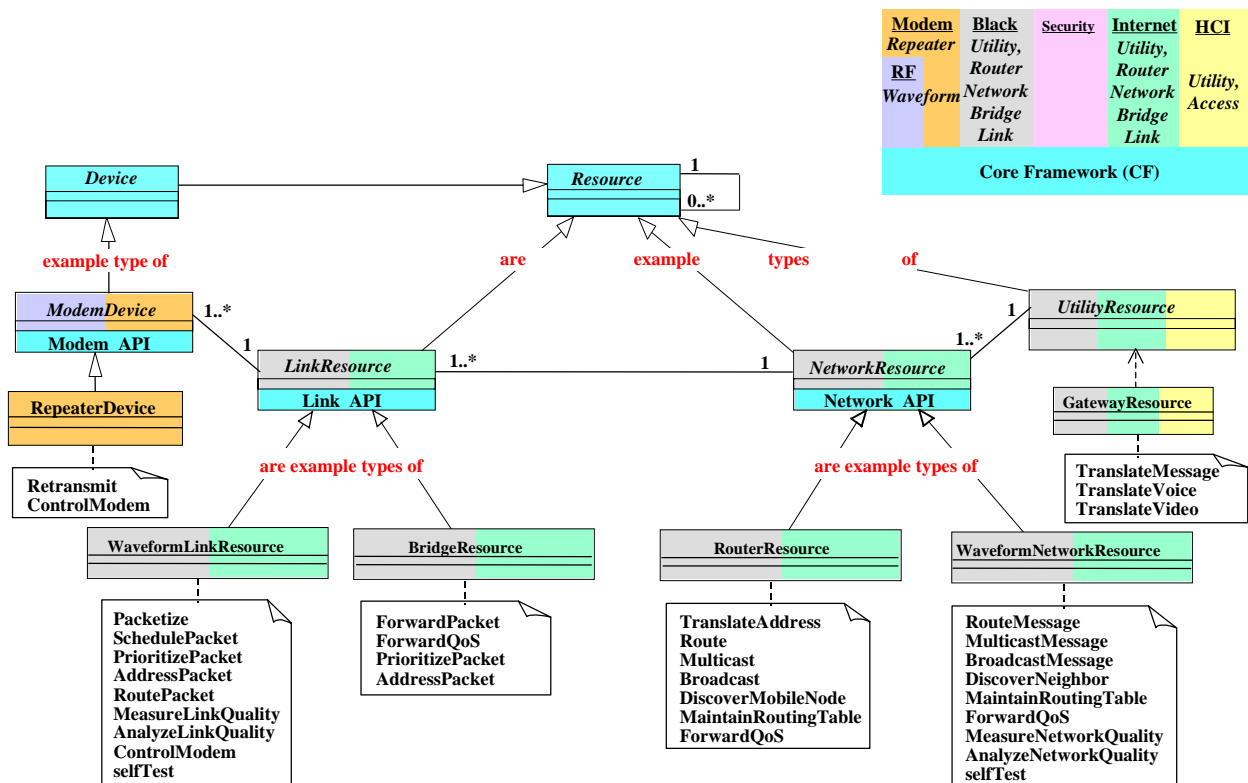


Figure 2-6. Example of Networking Resources



#### 2.2.1.7.4 I/ODevice Functionality.

Examples of *I/ODevices* are shown in figure 2-7. An *I/ODevice* provides access to system hardware devices and external physical interfaces. The operations performed by an *I/ODevice* will vary depending on the system hardware assets as well as the physical interfaces to be supported and are not dictated by the CF. Typical I/O operations are depicted within the example subclasses.

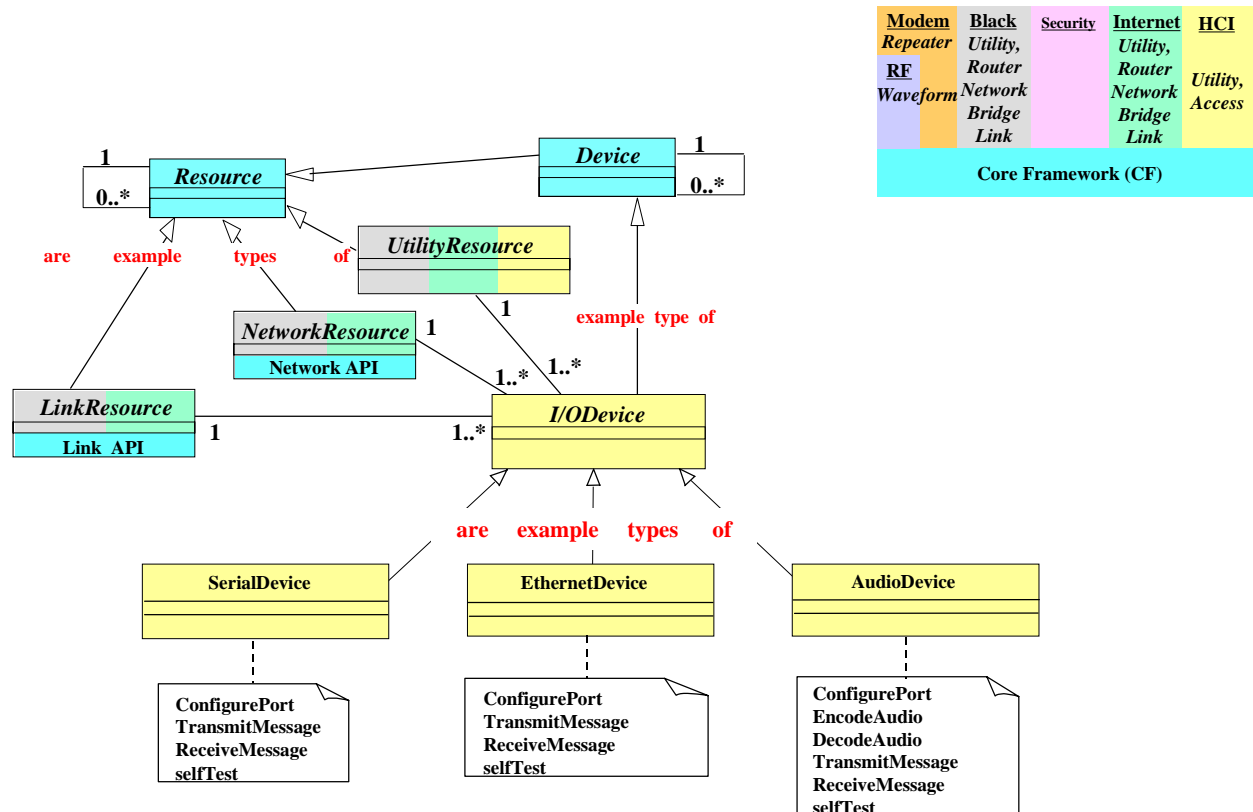


Figure 2-7. Examples of I/O Resources

### 2.2.1.7.5 SecurityDevice Functionality.

Examples of *SecurityDevice* and *SecurityResource* are shown in figure 2-8. Typical security operations are depicted within the example subclasses. *SecurityDevice* subclasses extend security functions to hardware devices within the system while *SecurityResource* subclasses extend security functions to software components. There can be a wide variation of security solutions both in hardware and software. Transmission security (TRANSEC) and communications security (COMSEC) requirements also vary between waveforms. The location of the security boundary with respect to networking requirements also varies between waveforms. The CF base application interfaces are extended to *SecurityResources* through Security APIs, which provide standard interfaces for control and communication between security devices and resources and application waveforms.

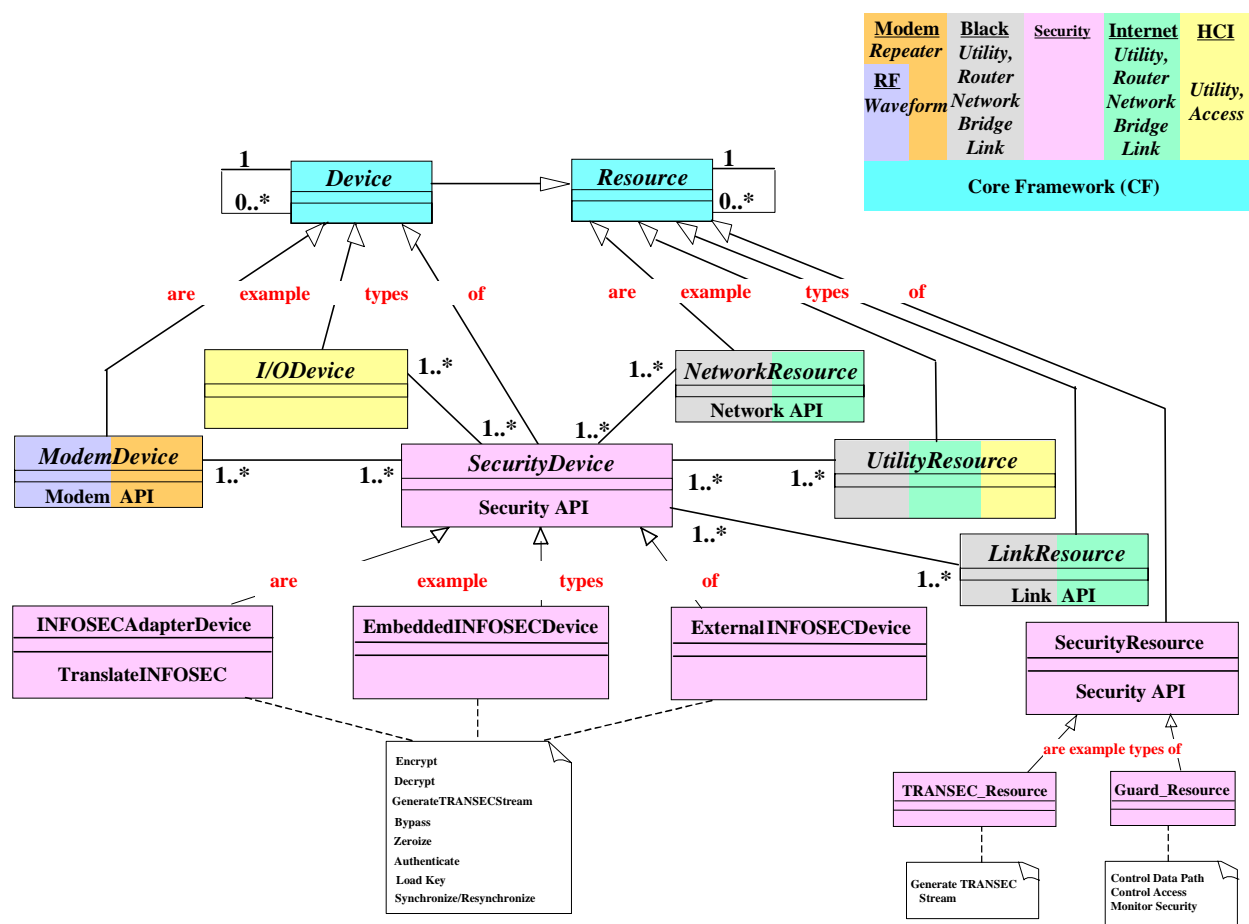


Figure 2-8. Examples of Security Devices and Resources

### 2.2.1.7.6 *UtilityResource* Functionality.

An example of *UtilityResource* is shown in Figure 2-9. The operations performed by the utility resources will vary depending on the embedded applications to be supported as well as host interface protocol requirements and are not dictated by the CF. Typical utility operations are depicted within the example subclasses. Ultimately, the *UtilityResource* encompasses any non-waveform application that could execute in an SCA-compliant system.

### 2.2.1.8 System Control.

The SCA provides a specification for interfaces, services, and data formats for the control of resources. Each resource establishes its controllable parameters with the *DomainManager* via a Domain Profile. Applications constrain each resource's parameter values to their own needs. Applications' controllable parameters are also in the Domain Profile.

Use of CORBA and the base application interfaces provides the means to have domain and application control through a common interface. *SerialDevice* and *EthernetDevice* (in Figure 2-7) are examples of the external interfaces available to a user. These examples show that system control operations operate with human or machine interfaces either locally or remotely and interact in a manner that facilitates portability.

Non-CORBA user terminals are interfaced through the use of Adapters.

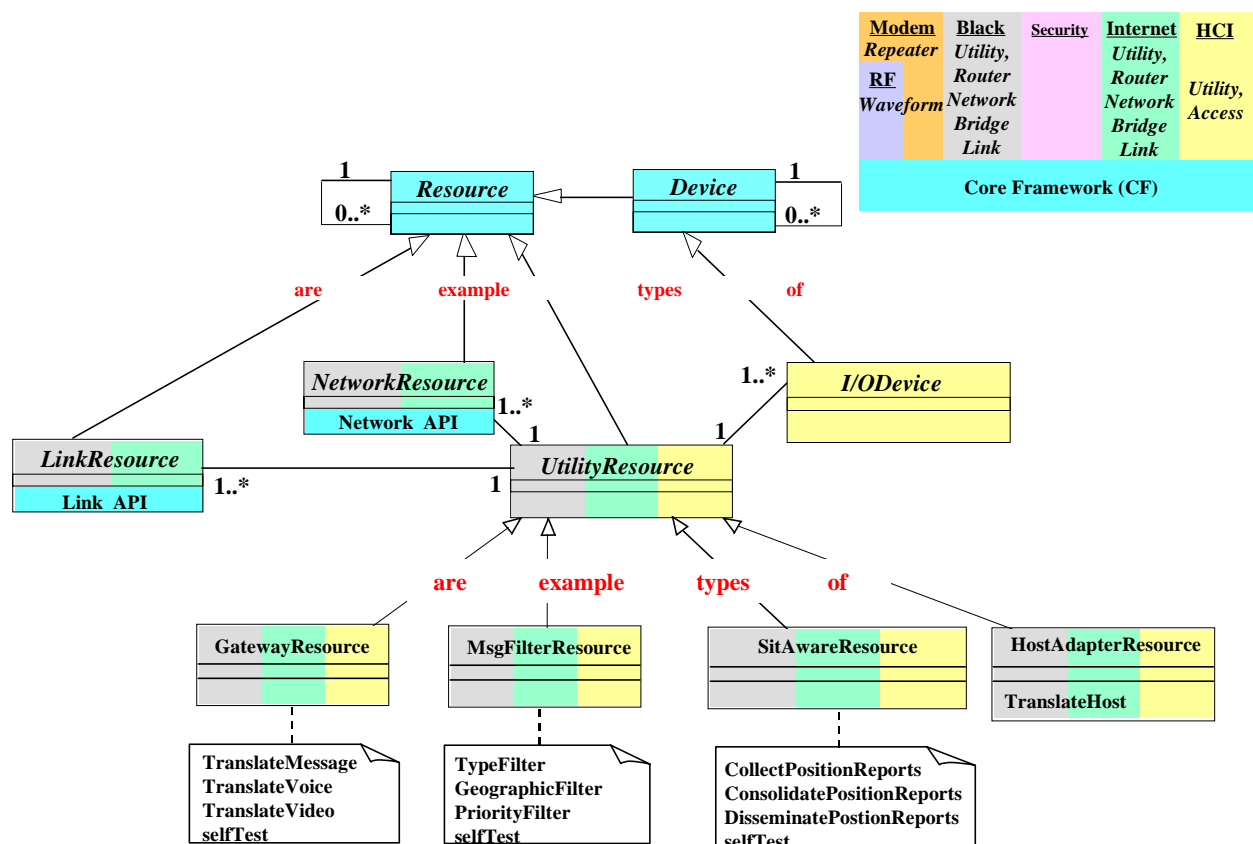
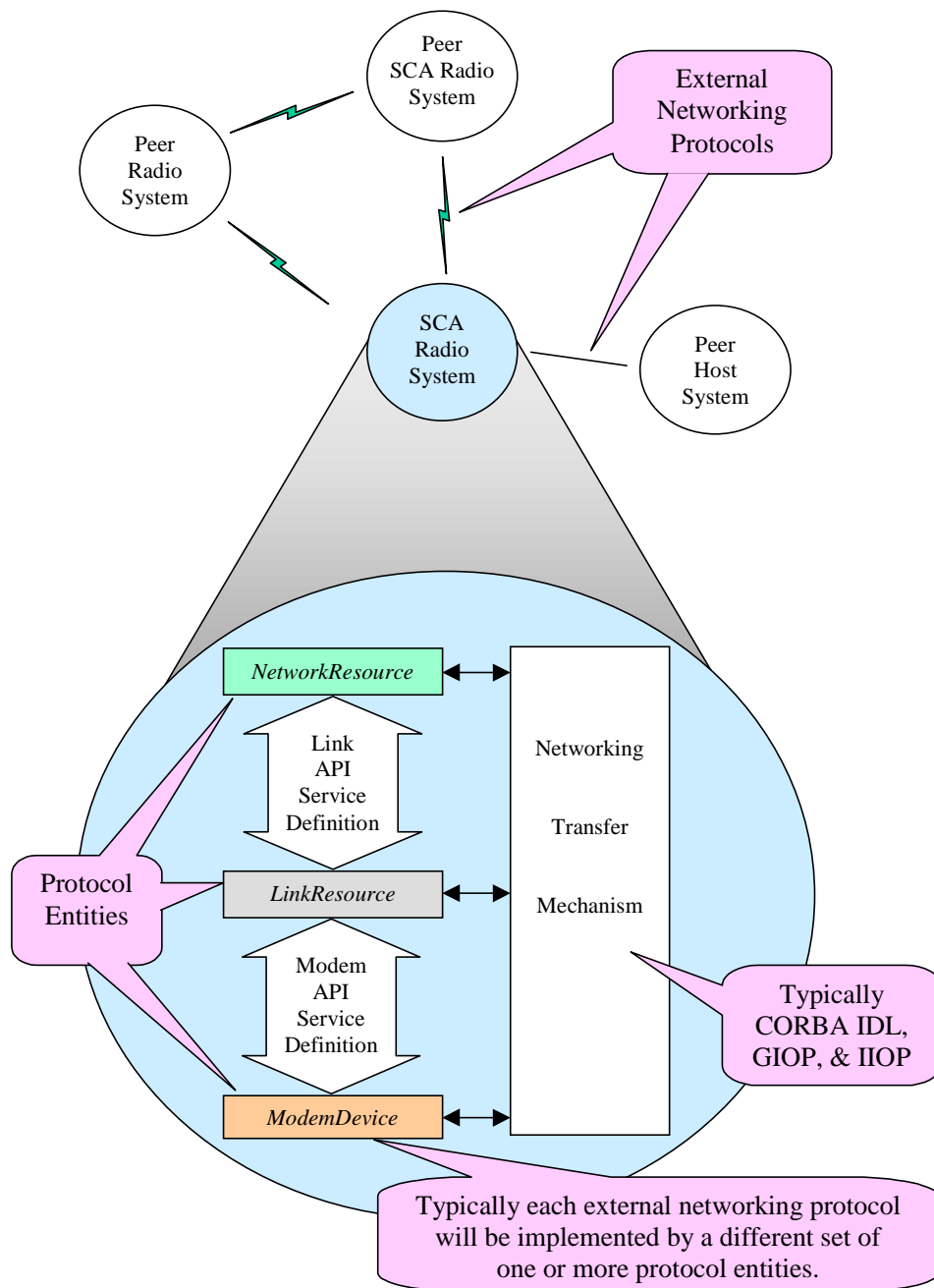


Figure 2-9. Example of Utility Resources

### 2.2.2 Networking Overview.

SCA-compliant Radio Systems communicate with peer systems through protocols as shown in figure 2-10. The external networking protocols between an SCA-compliant System and its peers are part of waveform applications and are not specified by this architecture specification. However, the interface definitions for the services required to implement the protocols within an SCA-compliant System are specified (in the API Supplement).



**Figure 2-10. External Network Protocols and SCA Support**

### 2.2.2.1 External Networking Protocols.

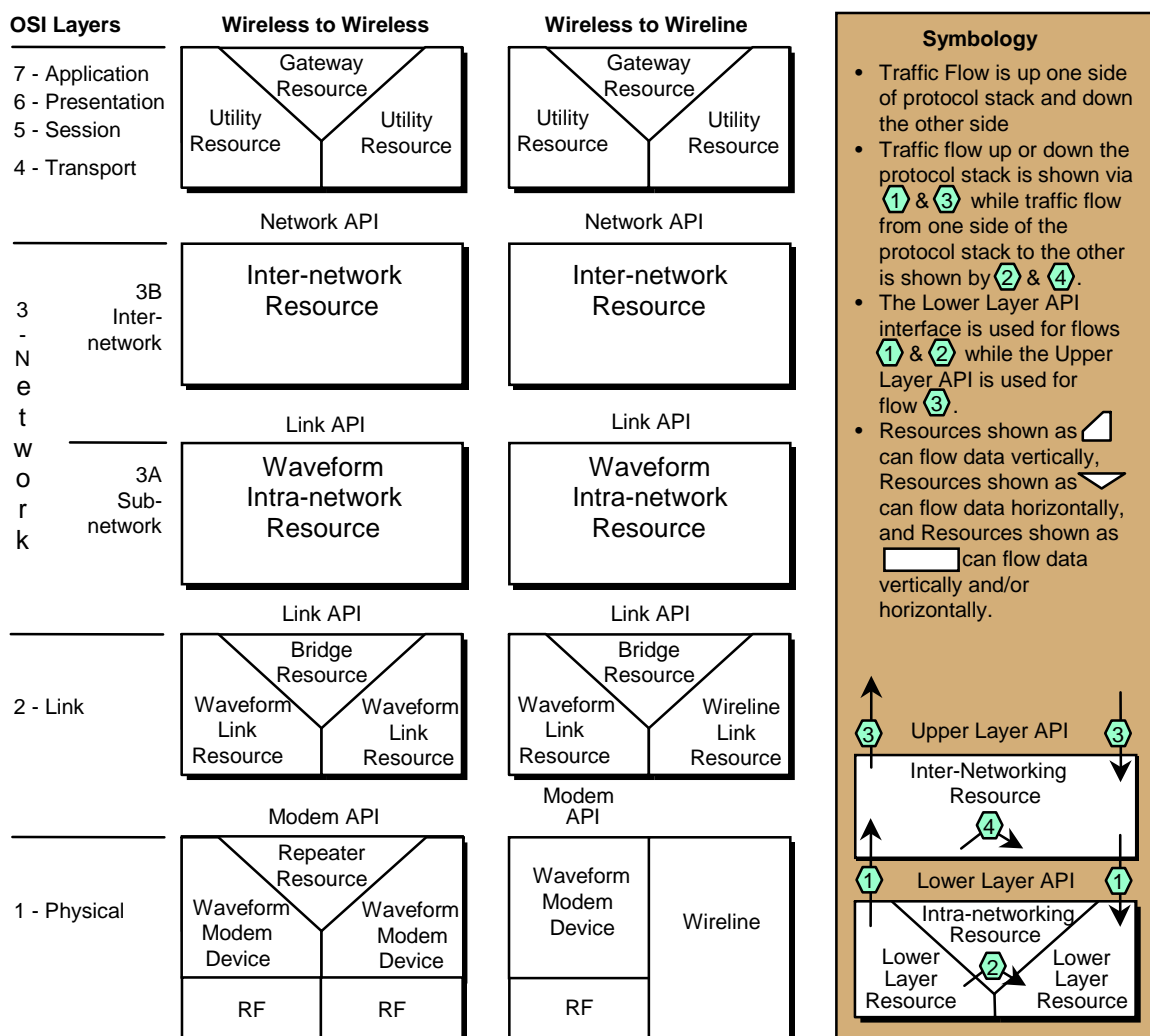
External networking protocols define the communications between an SCA-compliant Radio System and its peer systems. These external networking protocols can run over wireless or wireline physical media. Example protocols include Single Channel Ground/Airborne Radio System (SINCGARS), Ethernet, HF Automatic Link Establishment (ALE), IEEE 802.11, IS-95A, IP, and future networking protocols.

Through the external networking protocols, implemented by applications in an SCA-compliant radio system and its peer systems, a network of nodes is formed interconnected by repeaters, bridges, routers, and/or gateways. As shown in figure 2-11, external networking protocols will typically interconnect at different layers using:

1. Physical layer interconnections with a repeater function,
2. Link layer interconnections with a bridge function,
3. Network layer interconnections with standard network routing, and/or
4. Upper layer interconnections with application gateways.

The different categories of interoperability are outlined below based upon the OSI Model. There may be multiple levels of interoperability within the same system on a waveform by waveform basis.

- A. Physical Layer Interoperability. The external networking protocols provide a compatible physical interface, including the signaling interface, but no higher layer processing. This level of interoperability is adequate for a simple bit by bit bridging or relay operation between two interfaces.
- B. Link Layer Interoperability. The external networking protocols provide link layer processing over all physical interfaces. This level of interoperability is adequate for allowing the radio to be used as transport and for allowing the radio to use another network as transport. Intelligent routing or switching decisions are limited to local layer 2 routing.
- C. Network Layer Interoperability. The external networking protocols provide network layer address processing interoperability. The radio and the networks being inter-operated are sub-networks of the same Inter-network. At this level, intelligent switching and routing decisions can be made end-to-end.
- D. Host Level Interoperability (Layers 4 – 7). Embedded applications can exchange information with hosts attached to the network. An example of this is a handheld radio that contains an embedded Situation Awareness (SA) application exchanging SA updates with a vehicular platform in an external sub-network. In this example, the radio provides message payload translations to allow two otherwise incompatible hosts to communicate.



**Figure 2-11. SCA-Supported Networking Mapped to OSI Network Model**

#### 2.2.2.2 SCA Support for External Networking Protocols.

Figure 2-10 shows that within an SCA-compliant Radio System, application protocol entities are used to implement the external networking protocols. These protocol entities are networking applications<sup>2</sup>. Entity types that support external networking protocols include *ModemDevice*, *LinkResource*, *NetworkResource*, *SecurityDevice*, *I/ODevice*, and *UtilityResource*. Typically,

<sup>2</sup> External networking protocol entities can reside within an application or within the kernel space of operating systems. These external networking protocol applications are not necessarily the same as OSI layer 7 applications. (When an application uses protocol entities within the OS kernel space, and that kernel space is also used for internal system CORBA transport protocol, additional security protection may be required to prevent external network nodes from directly connecting with internal CORBA objects.)

each waveform or wireline protocol will be implemented by a unique set of one or more protocol entities. A unique set of protocol entities implements the protocol stack specified by a waveform or wireline protocol. A radio system implementing multiple waveform applications may have multiple protocol entities at each protocol layer.

In order to support application portability, standard interfaces are required between application protocol entities. These Networking APIs, support the concept of a service interface between a service provider (usually the lower OSI protocol layer) and a service user (usually the higher OSI protocol layer).

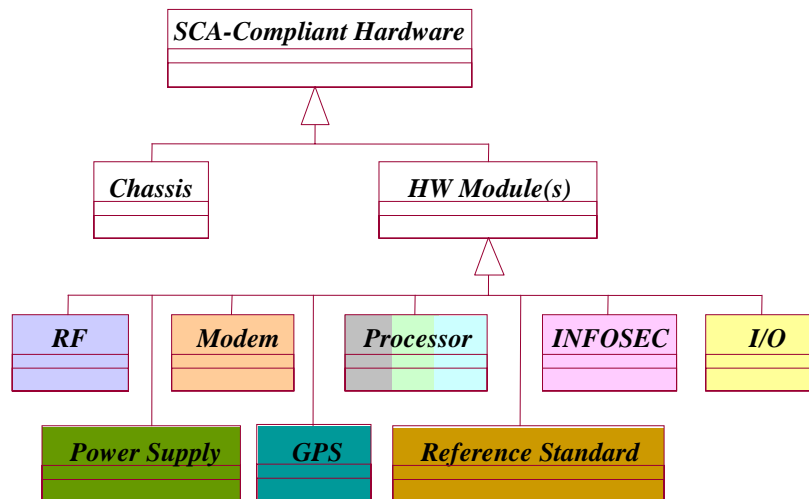
Networking APIs, like other waveform application APIs, are extensions to the CF base application interfaces that are inherited from the *Resource* class. APIs can be extended allowing vendors to provide value-added features that distinguish themselves from their competitors.

Two Networking API types are illustrated in this section: a Link API associated with the *LinkResource* and a Network API associated with the *NetworkResource*. (A Modem API is associated with the *ModemDevice* for a Networking OSI link layer protocol as well as for other, non-networking applications.) The APIs can be mapped into the OSI Networking Protocol model as shown in figure 2-11. This figure shows two very similar protocol stacks for wireless-to-wireless networking and wireless-to-wireline networking. The difference is that the wireline stack has a *WirelineDevice* at the physical layer instead of a *ModemDevice*. (Note that the OSI network layer maybe split into multiple network resources as shown in figure 2-11. In most cases, the layer 3A sub-network has a link API to the upper layer 3B inter-network (for example when layer 3B is IP). However, for some network waveform protocols, the layer 3A interface may be the network API).

The SCA defines a Networking API Instance to provide the mechanism for distributing the protocol layers within a SCA-compliant Radio System. A Networking API Instance is a coupling of a Networking API Service Definition and a Networking Transfer Mechanism for a particular waveform implementation. The Service Definition for a waveform protocol layer details the primitives (operations), the parameters (variables), their representation (structures, types, formats), and its behavior. The networking transfer mechanism provides the communication between the waveform protocol layer service provider and a service user. CORBA is the preferred transfer mechanism. Because security requirements for a particular implementation may be met using services associated with CORBA, later introduction of a different transfer mechanism requires careful analysis of the security services that can be provided by that transfer mechanism. Figure 2-10 shows the relationship between protocol entities, Service Definitions, and Networking Transfer Mechanisms.

### 2.2.3 Overview - Hardware Architecture.

Partitioning the hardware into classes places emphasis on the physical elements of the system and how they are composed of functional elements. These classes define common elements sharing physical attributes (characteristics and interfaces) that carry over to implementation for specific domain platforms. The same framework applies to all domains. Appropriate application of the requirements leads to common hardware modules for different platforms. A summary view of the hardware framework is shown in figure 2-12.



**Figure 2-12. Hardware Architecture Framework**

The *HWModule(s)* class inherits the system level attributes from the *SCA-CompliantHardware* class. Classes below the *HWModule(s)* class inherit the attributes of that class. The attributes are the parameters that define domain-neutral hardware devices, and the values assigned to the attributes satisfy requirements for a selected implementation. The hardware devices, which are the physical implementation of these classes, will have values for the relevant attributes based on a platform's physical requirements and the procurement performance requirements. Some attributes are used in the creation of waveform applications and provided in a Device Profile, readable by CF applications.

The Chassis Class has unique physical, interface, platform power, and external environment attributes that are not shared with the modules in the chassis.



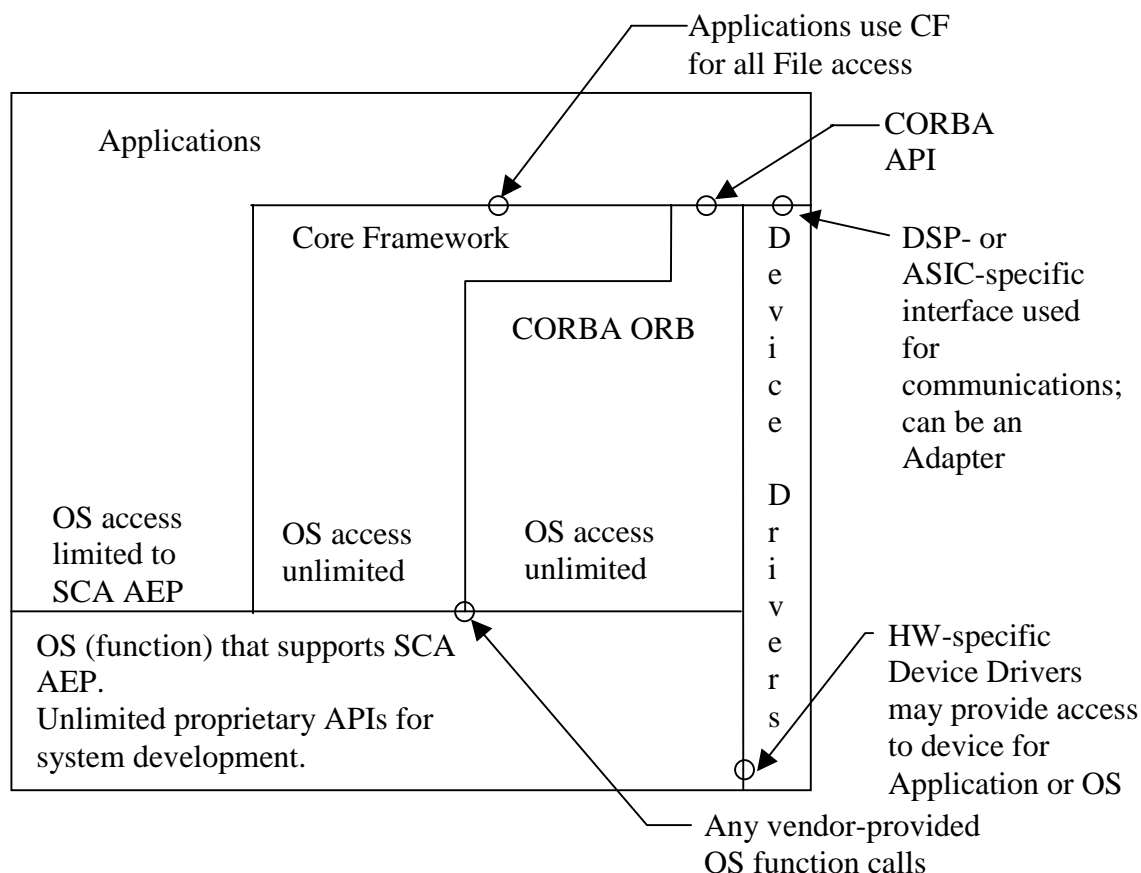
### 3 SOFTWARE ARCHITECTURE DEFINITION

#### 3.1 OPERATING ENVIRONMENT.

This section contains the requirements of the operating system, middleware, and the CF interfaces and operations that comprise the OE.

##### 3.1.1 Operating System.

The processing environment and the functions performed in the architecture impose differing constraints on the architecture. An SCA application environment profile (AEP) was defined to support portability of waveforms, scalability of the architecture, and commercial viability. POSIX services are used as a basis for this profile. The notional relationship of the OE and applications to the SCA AEP is depicted in figure 3-1. The OS shall provide the services designated as mandatory by the AEP defined in Appendix B. The OS is not limited to providing the services designated as mandatory by the profile. The CORBA Object Request Broker (ORB) and the CF are not limited to using the services designated as mandatory by the profile.



**Figure 3-1. Notional Relationship of OE and Application to the SCA AEP**

Applications are limited to using the OS services that are designated as mandatory for the profile. Applications will perform file access through the CF. (Application requirements are covered in section 3.2.)

### 3.1.2 Middleware & Services.

#### 3.1.2.1 CORBA.

The OE shall contain CORBA middleware and services. At a minimum, the ORB and related software shall comply with Minimum CORBA as specified by the OMG Document orbos/98-05-13, May 19, 1998.

#### 3.1.2.2 CORBA Extensions.

No extensions and/or services above and beyond Minimum CORBA shall be used except as specifically identified below.

*{The listed extensions are considered important for the architecture to meet its goals; however, they are not commercially available at the time of this release. When they become readily available, they will be reconsidered for mandatory inclusion.}*

##### 3.1.2.2.1 Naming Service.

CORBA Naming Service may be used. If a CORBA Naming Service is used, the OE should provide an Interoperable Naming Service as specified by the OMG Document orbos/98-10-11, October 19, 1998.

As an alternative, software components will include stringified Interoperable Object References (IORs) in their Software Profile.

##### 3.1.2.2.2 Quality of Service Control.

###### 3.1.2.2.2.1 Real-Time.

The OE should provide the Real-Time CORBA extension as specified by the OMG Document orbos/98-12-05, December 21, 1998.

###### 3.1.2.2.2.2 Messaging.

CORBA Messaging as specified by the OMG Document orbos/98-05-05, May 18, 1998 should be provided in order to provide the policy framework used by the Real-Time CORBA extension.

*{additional QoS requirements and policies are under evaluation.}*

### 3.1.3 Core Framework.

The CF specification includes a detailed description of the purpose of each interface, the purpose of each supported operation within the interface, and interface class diagrams to support these descriptions. The corresponding IDL for the CF can be found in Appendix C.

Figure 3-2 depicts the key elements of the CF and the relationships between these elements. A *DomainManager* component manages the software *Applications*, *ApplicationFactories*, and hardware devices (*Devices* and *DeviceManagers*) within the system. An *Application* is a type of *Resource* and consists of one to many software *Resources*. Some of the software *Resources* may directly control the system's internal hardware devices; these *Resources* are logical *Devices*. (For example, a *ModemDevice* may provide direct control of a modem hardware device such as a Field Programmable Gate Array (FPGA) or an Application Specific Integrated Circuit (ASIC).

The diagram illustrates the OE architecture with the following components and relationships:

- Non-Core Applications** (purple box)
- Core Framework (CF)** (cyan box)
- Commercial Off-the-Shelf (COTS)** (blue box)

**Core Framework (CF) Components:**

- Port** (purple box)
- LifeCycle** (purple box)
- PropertySet** (purple box)
- TestableObject** (purple box)
- Resource** (purple box)
- ResourceFactory** (purple box)
- Application** (purple box)
- ApplicationFactory** (purple box)
- Device** (purple box)
- DomainManager** (purple box)
- File** (cyan box)
- FileSystem** (cyan box)
- FileManager** (cyan box)

**Relationships:**

- Port** is a base class for **LifeCycle**, **PropertySet**, and **TestableObject**.
- LifeCycle** inherits from **PropertySet**.
- TestableObject** inherits from **Resource**.
- Resource** inherits from **Application**.
- ResourceFactory** uses **Resource**.
- ApplicationFactory** uses **Application**.
- Device** is a base class for **DomainManager**.
- DomainManager** uses **Device** (0..\* devices).
- DomainManager** uses **Application** (0..\* applications).
- DomainManager** uses **Resource** (0..\* resources).
- DomainManager** uses **File** (1..\* fileMgr).
- DomainManager** uses **FileSystem** (1..\* fileMgr).
- DomainManager** uses **FileManager** (1..\* fileMgr).
- DeviceManager** uses **Device** (0..\* devices).
- DeviceManager** uses **DomainManager** (1..\* deviceManagers).
- DeviceManager** uses **File** (1..\* fileMgr).
- DeviceManager** uses **FileSystem** (1..\* fileMgr).
- DeviceManager** uses **FileManager** (1..\* fileMgr).
- StringConsumer** uses **Logger** (dashed arrow).
- Logger** uses **DeviceManager** (dashed arrow).
- DeviceManager** uses **Device** (dashed arrow).
- DeviceManager** uses **DomainManager** (dashed arrow).
- DeviceManager** uses **File** (dashed arrow).
- DeviceManager** uses **FileSystem** (dashed arrow).
- DeviceManager** uses **FileManager** (dashed arrow).
- DomainManager** uses **Device** (dashed arrow).
- DomainManager** uses **Application** (dashed arrow).
- DomainManager** uses **Resource** (dashed arrow).
- DomainManager** uses **File** (dashed arrow).
- DomainManager** uses **FileSystem** (dashed arrow).
- DomainManager** uses **FileManager** (dashed arrow).
- File** is a base class for **FileSystem**.
- FileSystem** is a base class for **FileManager**.

managed by the *DomainManager* are CORBA objects. Some *Resources* may be dependent on other *Resources*. The way of creating up and tearing down any *Resource* will be created by using a *ResourceFactory* interface or by using the *FileManager*, *FileSystem*, and *File* interfaces. These interfaces are used for creating files within the system, and for loading and unloading files that the *Devices* execute upon.

The file service interfaces (*FileManager*, *FileSystem*, and *File*) are used for installation and removal of application files within the system, and for loading and unloading application files on the various processors that the *Devices* execute upon.

If user security controls are required, then the CORBA context capability shall be used for setting the user access authorities that are passed along with the CF call. This context information is used by the CF implementation for verifying user authorization before the CF operation is performed. The context content and format is implementation specific. In addition, the file verification mechanism to be used is implementation specific.

### 3.1.3.1 Base Application Interfaces.

#### 3.1.3.1.1 *Port*.

##### 3.1.3.1.1.1 Description.

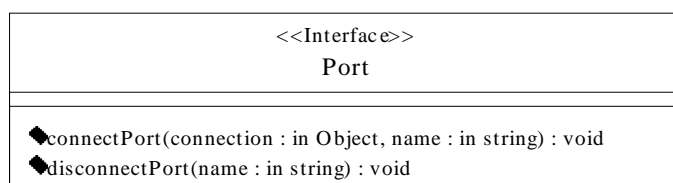
This interface provides operations for managing associations between ports. An application defines a specific *Port* type by specifying an interface that inherits the *Port* interface. An application establishes the operations for transferring data and control. The application also establishes the meaning of the data and control values. Whether a particular operation is push or pull, synchronous or asynchronous, or mono- or bi-directional or uses flow control (e.g., pause, start, stop), is application-dependent.

The nature of *Port* fan-in, fan-out, or one-to-one is component dependent.

Note 1: The CORBA specification defines only a minimum size for each basic IDL type. The actual size of the data type is dependent on the language (defined in the language mappings) as well as the Central Processing Unit (CPU) architecture used. By using these CORBA basic data types, portability is maintained between components implemented in differing CPU architectures and languages.

Note 2: How components' ports are connected is described in the software assembly descriptor file of the Domain Profile (3.1.3.4).

##### 3.1.3.1.1.2 UML.



**Figure 3-3. *Port* Interface UML**

##### 3.1.3.1.1.3 Types.

###### 3.1.3.1.1.3.1 InvalidPort.

```
exception InvalidPort { unsigned short errorCode, string msg };
```

This exception indicates one of the following errors has occurred in the specification of a *Port* association:

- ErrorCode 1 means the *Port* component is invalid (unable to narrow object reference) or illegal object reference,
- ErrorCode 2 means the *Port* name is not found (not used by this *Port*)

### 3.1.3.1.1.3.2 OccupiedPort.

```
exception OccupiedPort {};
```

This exception indicates the Port is unable to accept any additional connections.

### 3.1.3.1.1.4 Attributes.

Not applicable (N/A).

### 3.1.3.1.1.5 Operations.

#### 3.1.3.1.1.5.1 *connectPort*.

##### 3.1.3.1.1.5.1.1 Brief Rationale.

Applications require the *connectPort* operation to establish associations between *Ports*. *Ports* provide channels through which data and/or control pass.

The *connectPort* operation provides half of a two-way association; therefore two calls are required to create a two-way association.

##### 3.1.3.1.1.5.1.2 Synopsis.

```
void connectPort(in Object connection, in string name) raises (InvalidPort,
OccupiedPort);
```

##### 3.1.3.1.1.5.1.3 Behavior.

The *connectPort* operation shall cause a consumer/producer component to be associated with its counterpart component. The *connectPort* operation establishes only half of the association. Name is used to clearly identify the type of *Port* that is being connected to this *Port*, so that the CORBA object reference can be narrowed to a specific IDL interface based on the name.

##### 3.1.3.1.1.5.1.4 Returns.

This operation does not return a value.

##### 3.1.3.1.1.5.1.5 Exceptions/Errors.

The InvalidPort exception shall be raised when the *Port* component passed to *connectPort* is not a valid *Port* component.

The OccupiedPort exception shall be raised when unable to accept the connections because the *Port* is already fully occupied.

#### 3.1.3.1.1.5.2 *disconnectPort*.

##### 3.1.3.1.1.5.2.1 Brief Rationale.

Applications require the *disconnectPort* operation in order to allow consumer/producer data components to disassociate themselves from their counterparts (consumer/producer).

##### 3.1.3.1.1.5.2.2 Synopsis.

```
void disconnectPort (in string name) raises (InvalidPort);
```

##### 3.1.3.1.1.5.2.3 Behavior.

The *disconnectPort* operation shall cause a consumer or producer component to be disassociated from its counterpart component.

#### 3.1.3.1.1.5.2.4 Returns.

This operation does not return a value.

#### 3.1.3.1.1.5.2.5 Exceptions/Errors.

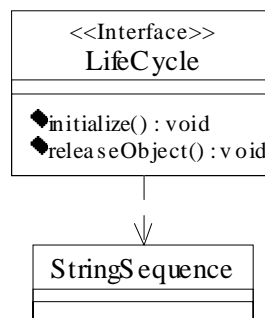
The `InvalidPort` exception shall be raised when the name passed to *disconnectPort* is not associated with the *Port* component.

### 3.1.3.1.2 LifeCycle.

#### 3.1.3.1.2.1 Description.

The *LifeCycle* interface defines the generic operations for initializing or releasing instantiated component-specific data and/or processing elements.

#### 3.1.3.1.2.2 UML.



**Figure 3-4. *LifeCycle* Interface UML**

#### 3.1.3.1.2.3 Types.

##### 3.1.3.1.2.3.1 InitializeError.

```
exception InitializeError { string message; };
```

This exception indicates an error occurred during component initialization. The message shall provide additional information describing the reason why the error occurred.

##### 3.1.3.1.2.3.2 ReleaseError.

```
exception ReleaseError { string message; };
```

This exception indicates an error occurred during component `releaseObject`. The message shall provide additional information describing the reason why the error occurred.

#### 3.1.3.1.2.4 Attributes.

N/A.

#### 3.1.3.1.2.5 Operations.

##### 3.1.3.1.2.5.1 *initialize*.

##### 3.1.3.1.2.5.1.1 Brief Rationale.

The purpose of the *initialize* operation is to provide a mechanism to set a component to a known initial state. (For example, data structures may be set to initial values, memory may be allocated, hardware components may be configured to some state, etc.)

#### 3.1.3.1.2.5.1.2 Synopsis.

```
void initialize() raises (InitializeError);
```

#### 3.1.3.1.2.5.1.3 Behavior.

The *initialize* operation shall be invoked one time for a component, at instantiation. Initialization behavior is implementation dependent.

#### 3.1.3.1.2.5.1.4 Returns.

This operation does not return a value.

#### 3.1.3.1.2.5.1.5 Exceptions/Errors.

The *InitializeError* exception shall be raised when an initialization error occurs.

#### 3.1.3.1.2.5.2 releaseObject.

##### 3.1.3.1.2.5.2.1 Brief Rationale.

The purpose of the *releaseObject* operation is to provide a means by which an instantiated component may be torn down.

##### 3.1.3.1.2.5.2.2 Synopsis.

```
void releaseObject() raises (ReleaseError);
```

##### 3.1.3.1.2.5.2.3 Behavior.

The behavior of this operation is implementation dependent. For example, if the instantiated component has been created via the *ResourceFactory* interface, the release operation could delegate to a *ResourceFactory* implementation in order to decrement the object reference count and ultimately tear down the instantiated component.

The component shall release all internal memory allocated during the instantiation.

##### 3.1.3.1.2.5.2.4 Returns.

This operation does not return a value.

##### 3.1.3.1.2.5.2.5 Exceptions/Errors.

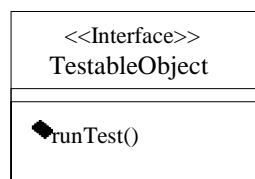
A *ReleaseError* exception shall be raised when a release error occurs.

#### 3.1.3.1.3 TestableObject.

##### 3.1.3.1.3.1 Description.

The *TestableObject* interface defines a set of operations that can be used to test component implementations.

##### 3.1.3.1.3.2 UML.



**Figure 3-5. *TestableObject* Interface UML**

### 3.1.3.1.3.3 Types.

#### 3.1.3.1.3.3.1 UnknownTest.

```
exception UnknownTest {};
```

This exception indicates the requested test to be performed is unknown by the component.

### 3.1.3.1.3.4 Attributes.

N/A.

### 3.1.3.1.3.5 Operations.

#### 3.1.3.1.3.5.1 *runTest*.

##### 3.1.3.1.3.5.1.1 Brief Rationale.

The *runTest* operation allows components to be “blackbox” tested. This allows Built-In Test (BIT) to be implemented as well as provides a mean to isolate faults (both software and hardware) within the system.

##### 3.1.3.1.3.5.1.2 Synopsis.

```
long runTest(in unsigned long testNum)
raises UnknownTest;
```

##### 3.1.3.1.3.5.1.3 Behavior.

The *runTest* operation shall use the testNum argument to specify the implementation-specific test to be run. Tests to be implemented by a component are component-dependent and are specified in the component’s software profile.

##### 3.1.3.1.3.5.1.4 Returns.

The *runTest* operation shall return result of test performed. The values returned and their meanings are described in the component’s software profile.

##### 3.1.3.1.3.5.1.5 Exceptions/Errors.

An UnknownTest exception shall be raised when no such test is known by the component.

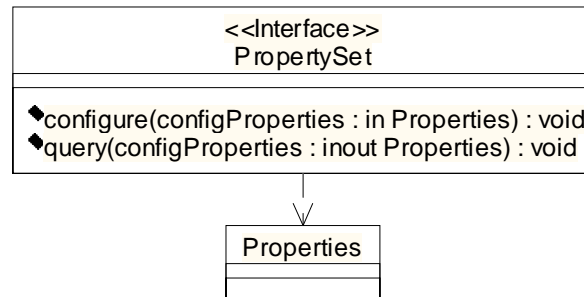
### 3.1.3.1.4 *PropertySet*.

#### 3.1.3.1.4.1 Description.

The *PropertySet* interface defines *configure* and *query* operations to access component properties/attributes.



## 3.1.3.1.4.2 UML.

**Figure 3-6. *PropertySet* Interface UML**

## 3.1.3.1.4.3 Types.

## 3.1.3.1.4.3.1 UnknownProperties.

```
exception UnknownProperties {properties invalidProperties;};
```

This exception indicates a set of properties unknown by the component.

## 3.1.3.1.4.3.2 InvalidConfiguration.

```
exception InvalidConfiguration { string msg; Properties invalidProperties};
```

This exception indicates the configuration of a Component has failed (no configuration at all was done). The message shall provide additional information describing the reason why the error occurred. The invalidProperties returned indicate the properties that were invalid.

## 3.1.3.1.4.3.3 PartialConfiguration.

```
exception PartialConfiguration { Properties invalidProperties};
```

This exception indicates the configuration of a Component was partially successful. The invalidProperties returned indicate the properties that were invalid.

## 3.1.3.1.4.4 Attributes.

N/A.

## 3.1.3.1.4.5 Operations.

3.1.3.1.4.5.1 *configure*.

## 3.1.3.1.4.5.1.1 Brief Rationale.

The purpose of this operation is to allow id/value pair configuration properties to be assigned to components implementing this interface.

## 3.1.3.1.4.5.1.2 Synopsis.

```
void configure(in Properties configProperties) raises (InvalidConfiguration,
PartialConfiguration);
```

## 3.1.3.1.4.5.1.3 Behavior.

The *configure* operation shall assign the id/value pair passed in the configProperties argument to the component.

#### 3.1.3.1.4.5.1.4 Returns.

This operation does not return a value.

#### 3.1.3.1.4.5.1.5 Exceptions/Errors.

An `InvalidConfiguration` exception shall be raised when a configuration error occurs preventing any property configuration on the component.

A `PartialConfiguration` exception shall be raised when some configuration properties were successful and some configuration properties were not successful.

#### 3.1.3.1.4.5.2 *query*.

##### 3.1.3.1.4.5.2.1 Brief Rationale.

The purpose of this operation is to allow a component to be queried to retrieve its properties.

##### 3.1.3.1.4.5.2.2 Synopsis.

```
void query(inout Properties configProperties) raises (UnknownProperties);
```

##### 3.1.3.1.4.5.2.3 Behavior.

The *query* operation shall retrieve a component's requested configuration data. If the `configProperties` are zero size then all component properties shall be returned. If the `configProperties` are not zero size then only those id/value pairs specified in the `configProperties` shall be returned.

#### 3.1.3.1.4.5.2.4 Returns.

This operation does not return a value.

#### 3.1.3.1.4.5.2.5 Exceptions/Errors.

The `UnknownProperties` exception shall be raised when one or more properties being requested are not known by the component.

#### 3.1.3.1.5 *Resource*.

##### 3.1.3.1.5.1 Description.

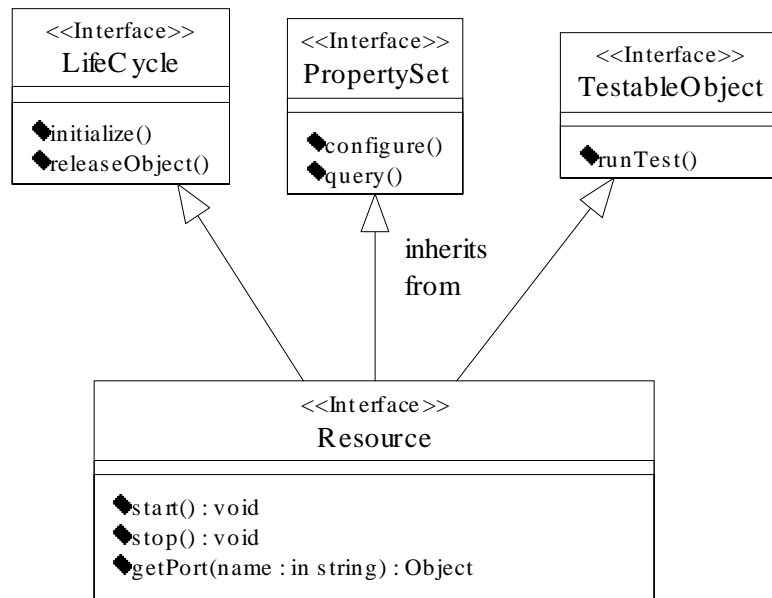
The *Resource* interface provides a common API for the control and configuration of a software component.

The *Resource* interface inherits from the *LifeCycle*, *PropertySet*, and *TestableObject* interfaces.

The inherited *LifeCycle*, *PropertySet*, and *TestableObject* interface operations are documented in their respective sections of this document.

The CF *Resource* interface may also be inherited by other application interfaces as described in the Software Profile's Software Component Descriptor (SCD) file (see 3.1.3.4).

## 3.1.3.1.5.2 UML.

**Figure 3-7. Resource Interface UML**

## 3.1.3.1.5.3 Types.

## 3.1.3.1.5.3.1 UnknownPort.

```
exception UnknownPort { };
```

This exception is raised if an undefined port is requested.

## 3.1.3.1.5.3.2 StartError.

```
exception StartError { string msg };
```

This exception indicates that an error occurred during an attempt to start the *Resource*. The message shall provide additional information describing the reason for the error and the severity of the error.

## 3.1.3.1.5.3.3 StopError.

```
exception StopError { string msg };
```

This exception indicates that an error occurred during an attempt to stop the *Resource*. The message shall provide additional information describing the reason for the error and the severity of the error.

## 3.1.3.1.5.4 Attributes.

N/A.

### 3.1.3.1.5.5 Operations.

#### 3.1.3.1.5.5.1 *stop*.

##### 3.1.3.1.5.5.1.1 Brief Rationale.

The *stop* operation is provided to command a *Resource* implementing this interface to stop internal processing.

##### 3.1.3.1.5.5.1.2 Synopsis.

```
void stop() raises (StopError);
```

##### 3.1.3.1.5.5.1.3 Behavior.

The *stop* operation shall disable all operations for the *Resource*.

##### 3.1.3.1.5.5.1.4 Returns.

This operation does not return a value.

##### 3.1.3.1.5.5.1.5 Exceptions/Errors.

The *StopError* exception shall be raised if an error occurs while stopping the resource.

#### 3.1.3.1.5.5.2 *start*.

##### 3.1.3.1.5.5.2.1 Brief Rationale.

The *start* operation is provided to command a *Resource* implementing this interface to start internal processing.

##### 3.1.3.1.5.5.2.2 Synopsis.

```
void start() raises (StartError);
```

##### 3.1.3.1.5.5.2.3 Behavior.

The *start* operation shall enable operations for the *Resource*.

##### 3.1.3.1.5.5.2.4 Returns.

This operation does not return a value.

##### 3.1.3.1.5.5.2.5 Exceptions/Errors.

The *StartError* exception shall be raised if an error occurs while stopping the resource.

#### 3.1.3.1.5.5.3 *getPort*.

##### 3.1.3.1.5.5.3.1 Brief Rationale.

The *getPort* operation provides a mechanism to obtain a specific consumer or producer *Port*. A *Resource* may contain zero to many consumer and producer port components. The exact number is specified in the component's Software Profile SCD (section 3.1.3.4). These *Ports* can be either push or pull types. Multiple input and/or output ports provide flexibility for *Applications* and *Resources* that must manage varying priority levels and categories of incoming and outgoing messages, provide multi-threaded message handling, or other special message processing.

##### 3.1.3.1.5.5.3.2 Synopsis.

```
Object getPort(in string name) raises ( UnknownPort );
```

### 3.1.3.1.5.5.3.3 Behavior.

The *getPort* operations shall return the object reference to the named port as stated in the *Resource's* SCD.

### 3.1.3.1.5.5.3.4 Returns.

The *getPort* operation shall return the CORBA object reference that matches the input port name.

### 3.1.3.1.5.5.3.5 Exceptions/Errors.

The *getPort* operation shall raise an UnknownPort exception if the port name is invalid.

### 3.1.3.1.5.5.4 *releaseObject*.

#### 3.1.3.1.5.5.4.1 Brief Rationale.

The *releaseObject* operation (of *LifeCycle*) causes the *Resource* to terminate execution and return allocated capabilities to the system. Before terminating, the *Resource* removes the message connectivity with its associated components (e. g., consumer *Ports*, producer *Ports*, *Loggers*, etc.) in the domain.

#### 3.1.3.1.5.5.4.2 Synopsis.

```
void releaseObject() raises (ReleaseError);
```

#### 3.1.3.1.5.5.4.3 Behavior.

The *Resource* shall release (e.g., CORBA Release) all ports that have been connected to the *Resource* ports.

The *Resource* shall release all internal memory allocated during the instantiation of the *Resource*.

#### 3.1.3.1.5.5.4.4 Returns.

This operation does not return a value.

#### 3.1.3.1.5.5.4.5 Exceptions/Errors.

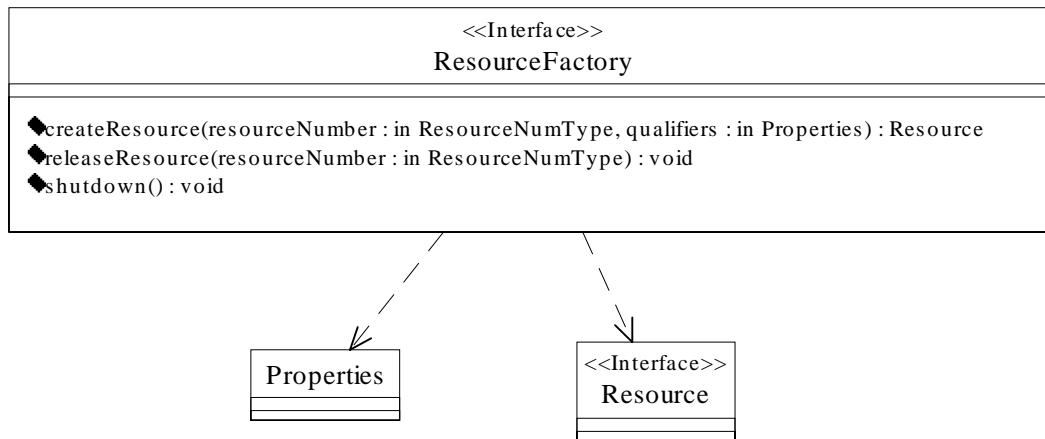
A ReleaseError shall be raised when the instantiated *Resource* incurs an error during the release process.

### 3.1.3.1.6 *ResourceFactory*.

#### 3.1.3.1.6.1 Description.

A *ResourceFactory* is used to create and tear down a *Resource*. The *ResourceFactory* interface is designed after the Factory Design Patterns. The factory mechanism provides client-server isolation among *Resources* (e.g., Network, Link, Modem, I/O, etc.) and provides an industry standard mechanism of obtaining a *Resource* without knowing its identity. An application is not required to use *ResourceFactories* to obtain, create, or tear down resources. A Software Profile will determine which application *ResourceFactories* are to be used by the *ApplicationFactory*.

## 3.1.3.1.6.2 UML.

Figure 3-8. *ResourceFactory* Interface UML

## 3.1.3.1.6.3 Types.

## 3.1.3.1.6.3.1 ResourceNumType.

This type defines the identity of a *Resource* created by the *ResourceFactory*.

```
typedef unsigned short ResourceNumType.
```

## 3.1.3.1.6.3.2 InvalidResourceNumber.

```
exception InvalidResourceNumber { string msg };
```

This exception indicates the resource number does not exist in the *Factory*. The message shall provide additional information describing why the resource number was invalid.

## 3.1.3.1.6.3.3 ShutdownFailure.

```
exception ShutdownFailure { string msg };
```

This exception indicates that the shutdown method failed to release the *ResourceFactory* from the CORBA environment due to the fact the *Factory* still contains *Resources*. The message shall provide additional information describing why the shutdown failed.

## 3.1.3.1.6.4 Attributes.

N/A.

## 3.1.3.1.6.5 Operations.

3.1.3.1.6.5.1 *createResource*.

## 3.1.3.1.6.5.1.1 Brief Rationale.

Applications may need to create *Resources* in another address space (e.g., process space, another processor, etc.) or without having the ability to directly create the *Resource* servant (e.g. the servant may be provided as part of the implementation of a commercial library).

### 3.1.3.1.6.5.1.2 Synopsis.

```
Resource createResource(in ResourceNumType resourceNumber, in Properties
qualifiers);
```

The resourceNumber is the identifier for *Resource*. The qualifiers will vary depending on the *ResourceFactory* implementation. The qualifiers passed by the *DomainManager* are described in the *Resource's* Software Profile.

### 3.1.3.1.6.5.1.3 Behavior.

If the *Resource* does not exist for the given resourceNumber, a *Resource* shall be created. A *ResourceFactory*, a *Resource*, or both may use the input qualifiers. The given resourceNumber shall be assigned to the new *Resource* and the reference count set to one. If the *Resource* already exists, that *Resource* is returned and the reference count shall be incremented by one.

### 3.1.3.1.6.5.1.4 Returns.

The *createResource* operation shall return a reference to a *Resource*. A nil CORBA component reference shall be returned if the operation is unable to create the resource.

### 3.1.3.1.6.5.1.5 Exceptions/Errors.

N/A.

### 3.1.3.1.6.5.2 *releaseResource*.

#### 3.1.3.1.6.5.2.1 Brief Rationale.

In CORBA there is client side and server side representation of a *Resource*. This operation provides the mechanism of releasing the *Resource* in the CORBA environment on the server side when all clients are through with a specific *Resource*. The client still has to release its client side reference of the *Resource*.

#### 3.1.3.1.6.5.2.2 Synopsis.

```
void releaseResource(in ResourceNumType resourceNumber) raises
{InvalidResourceNumber};
```

#### 3.1.3.1.6.5.2.3 Behavior.

For the specified resource, as indicated by the resourceNumber, the reference count shall be decremented. The *Resource* shall be torn down and released from the CORBA environment when the reference count is zero.

#### 3.1.3.1.6.5.2.4 Returns.

This operation does not return a value.

#### 3.1.3.1.6.5.2.5 Exceptions/Errors.

The InvalidResourceNumber exception shall be raised if an invalid resourceNumber is received.

### 3.1.3.1.6.5.3 *shutdown*.

#### 3.1.3.1.6.5.3.1 Brief Rationale.

In CORBA there is client side and server side representation of a *ResourceFactory*. This operation provides the mechanism for releasing the *ResourceFactory* from the CORBA environment on the server side. The client has the responsibility to release its client side reference of the *ResourceFactory*.

### 3.1.3.1.6.5.3.2 Synopsis.

```
void shutdown() raises {ShutdownFailure};
```

### 3.1.3.1.6.5.3.3 Behavior.

The *shutdown* operation shall cause the *ResourceFactory* to be torn down and released from the CORBA environment when no Resources exist in the *ResourceFactory*.

### 3.1.3.1.6.5.3.4 Returns.

This operation does not return a value.

### 3.1.3.1.6.5.3.5 Exceptions/Errors.

N/A.

## 3.1.3.2 Framework Control Interfaces.

Framework control within a Domain is accomplished by five interfaces: *Application*, *ApplicationFactory*, *DomainManager*, *DeviceManager*, and *Device*. The implementation of the *Application*, *ApplicationFactory*, and *DomainManager* interfaces are coupled together and must be delivered together as a complete domain management implementation. The *DeviceManager* and *Device* interfaces are used to manage devices in the domain. *DeviceManager* and *Device* can be implemented together by the same vendor or separately by different vendors. Framework Control Interfaces shall be implemented using the CF IDL presented in Appendix C.

### 3.1.3.2.1 *Application*.

#### 3.1.3.2.1.1 Description.

The *Application* class provides the interface for the control, configuration, and status of an instantiated *Application* in the domain.

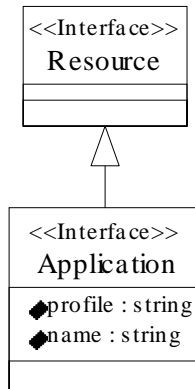
The *Application* interface class inherits the IDL interface of CF *Resource*. A created application instance may contain CF *Resource* components and/or non-CORBA components. If the created *Application* contains CF *Resource* components, the *PropertySet* and *TestableObject* operations are delegated to the HCI *Ports* contained within the application instance.

The *Application* interface *releaseObject* operation provides the interface to release the computing resources allocated during the instantiation of the *Application*, and de-allocate the devices associated with *Application* instance.

An instance of an *Application* is returned by the *create* operation of an instance of the *ApplicationFactory* class.



### 3.1.3.2.1.2 UML.



**Figure 3-9. Application Interface UML**

### 3.1.3.2.1.3 Types.

N/A.

### 3.1.3.2.1.4 Attributes.

#### 3.1.3.2.1.4.1 profile.

This profile attribute contains the Software Profile (3.1.3.4). CORBA-capable and non-CORBA-capable components have Profile files.

```
readonly attribute string profile;
```

#### 3.1.3.2.1.4.2 name.

This name attribute contains the name of the created *Application*. The *ApplicationFactory* interface's *create* operation name parameter provides the name content.

```
readonly attribute string name;
```

### 3.1.3.2.1.5 General Class Behavior.

The *Application* shall delegate the implementation of the inherited CF *Resource* operations, except *releaseObject*, to the *Application*'s *Resource* component(s) of the *Application* instance. The *getPort* operation shall return the *Application*'s ports on the Software Profile(s).

### 3.1.3.2.1.6 Operations.

#### 3.1.3.2.1.6.1 *releaseObject*.

##### 3.1.3.2.1.6.1.1 Brief Rationale.

The *releaseObject* operation terminates execution of the *Application*, returns all allocated computing resources, and de-allocates the devices associated with *Application*. Before terminating, the *Application* removes the message connectivity with its associated *Applications* (e. g., *Ports*, *Resources*, and *Loggers*) in the domain.

##### 3.1.3.2.1.6.1.2 Synopsis.

```
void releaseObject() raises (ReleaseError);
```

### 3.1.3.2.1.6.1.3 Behavior.

For each *Application* component, the *releaseObject* operation shall release the component by utilizing the *Resource's releaseObject* operation. If the component was created by a *ResourceFactory* component, the *Resource* shall be released utilizing the *ResourceFactory releaseResource* operation. The *ResourceFactory* components shall be shutdown utilizing the *ResourceFactory shutdown* operation.

For each allocated *Device*, the *releaseObject* operation shall terminate all processes / tasks of the *Application* components utilizing the *Device's terminate* operation.

For each allocated *Device*, the *releaseObject* operation shall de-allocate the memory associated with *Application* component instances utilizing the *Device's unload* operation.

The *releaseObject* operation shall de-allocate any internal *Application* memory associated with the *Application* instance.

The *releaseObject* operation shall de-allocate the *Devices* that are associated with the *Application* being released. The actual devices deallocated (*Device::deallocateCapacity*) shall reflect changes in capacity based upon component capacity requirements deallocated from them, which may also cause state changes for the *Devices*.

The *Application* shall release all client component references to the *Application* components.

The *Application* shall release all component references to itself maintained by the *Application's ResourceFactory*.

*Ports* shall be disconnected from other *Ports* that have been connected based upon the software profile.

For components (e.g., *Resource*, *ResourceFactory*) that are registered with Naming Service, the *releaseObject* operation shall unregister those components from Naming Service.

If security access control is being used in the system, then the *releaseObject* operation shall remove the *Application's Ports'* access setups in the access control database.

The *Application* shall, prior to successful *Application* release, log an Administrative event with the *DomainManager's Logger*.

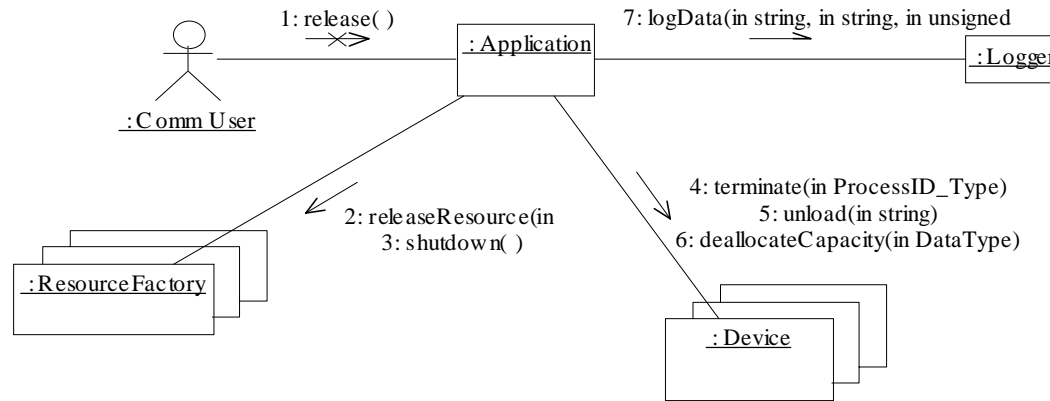
The *Application* shall, upon unsuccessful *Application* release, log an Alarm event with the *DomainManager's Logger*.

The following steps demonstrate one scenario of the *Application's* behavior for the release of an *Application* that contains *ResourceFactory* behavior:

1. Client invokes release operation.
2. Release the *Application* components.
3. Shutdown the *ResourceFactory* components.
4. Terminate the component's processes.
5. Unload the component's executable images.

6. Change the state of the associated device entries in the Domain Profile to be available, along with device(s) memory utilization availability and processor utilization availability based upon the Device Profiles and Software Profile.
7. Log an Event indicating that the *Application* was either successfully or unsuccessfully released.

Figure 3-10 is a collaboration diagram depicting the behavior as described above.



**Figure 3-10. Application Behavior**

#### 3.1.3.2.1.6.1.4 Returns.

This operation does not return a value.

#### 3.1.3.2.1.6.1.5 Exceptions/Errors.

A *ReleaseError* exception shall be raised when the *releaseObject* operation unsuccessfully releases the *Application* components due to internal processing errors.

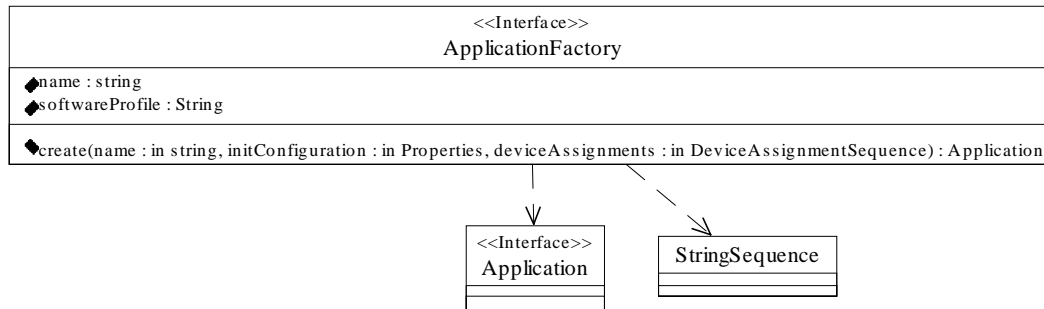
### 3.1.3.2.2 ApplicationFactory.

#### 3.1.3.2.2.1 Description

The *ApplicationFactory* interface class provides an interface to request the creation of a specific type of *Application* in the domain.

The *ApplicationFactory* interface class is designed using the Factory Design Pattern. The Software Profile determines the type of *Application* that is created by the *ApplicationFactory*.

## 3.1.3.2.2.2 UML.

Figure 3-11. *ApplicationFactory* UML

## 3.1.3.2.2.3 Types.

## 3.1.3.2.2.3.1 DeviceAssignmentType.

DeviceAssignmentType defines a structure that associates a component with the *Device* upon which the component must execute.

```

Struct DeviceAssignmentType
{
    string                componentID
    string                assignedDeviceID
}
  
```

## 3.1.3.2.2.3.2 DeviceAssignmentSequence.

The IDL sequence, DeviceAssignmentSequence, provides a unbounded sequence of 0..n of DeviceAssignmentType.

```

Typedef sequence<DeviceAssignmentType>DeviceAssignmentSequence;
  
```

## 3.1.3.2.2.3.3 exception CreateApplicationRequestError.

```

exception CreateApplicationRequestError
{
    DeviceAssignmentSequence invalidAssignment;
}
  
```

This exception is raised when the parameter DeviceAssignmentSequence contains one (1) or more invalid *Application* component-to-device assignment(s).

## 3.1.3.2.2.3.4 exception CreateApplicationError.

```

exception CreateApplicationError
{
    StringSequence          errorMessages;
}
  
```

This exception is raised when the *create* request is valid but the *Application* is unsuccessfully instantiated due to internal processing errors.

#### 3.1.3.2.2.4 Attributes.

##### 3.1.3.2.2.4.1 name.

The name attribute identifies the type of *Application* that can be instantiated by the *ApplicationFactory*.

```
readonly attribute string name;
```

##### 3.1.3.2.2.4.2 SoftwareProfile.

The SoftwareProfile attribute contains the Software Profile for the *Application* that can be created by the *ApplicationFactory*.

```
readonly attribute string softwareProfile;
```

#### 3.1.3.2.2.5 Operations.

##### 3.1.3.2.2.5.1 create.

###### 3.1.3.2.2.5.1.1 Brief Rationale.

This operation is used to create an *Application* within the system domain.

The *create* operation provides a client interface to request the creation of an *Application* on client requested device(s) or the creation of an *Application* in which the *ApplicationFactory* determines the necessary device(s) required for instantiation of the *Application*.

###### 3.1.3.2.2.5.1.2 Synopsis.

```
Application create(in string name, in Properties initConfiguration, in
DeviceAssignmentSequence deviceAssignments) raises ( CreateApplicationError,
CreateApplicationRequestError );
```

###### 3.1.3.2.2.5.1.3 Behavior.

If the input parameter deviceAssignments sequence length is zero (0), the *ApplicationFactory* shall determine the necessary devices to allocate for the creation of the *Application* using the Software Profile's software assembly descriptor (SAD). Each application will have a SAD.

An *Application* can be comprised of one or more components (e.g., *Resources*, *Devices*, etc.). The SAD contains Software Package Descriptors (SPDs) for each *Application* component. The SPD specifies the *Device* implementation criteria for loading dependencies (processor kind, etc.) and processing capacities (e.g., memory, process for an application component. The *create* operation shall use the SAD SPD implementation element to locate candidate devices capable of loading and executing *Application* components.

If deviceAssignments (not zero length) are provided, the *ApplicationFactory* shall verify each device assignment, for the specified component, against the component's SPD implementation element.

The *create* operation shall allocate (*Device::allocateCapacity*) component capacity requirements against candidate devices to determine which candidate devices satisfy all SPD implementation criteria requirements and SAD partitioning requirements (e.g, components HostCollocation, etc.). Only devices that have been granted successful capacity allocations shall be used for loading and executing *Application* components, or used for data processing. The actual devices chosen shall reflect changes in capacity based upon component capacity requirements allocated to them, which may also cause state changes for the *Devices*.

The *create* operation shall load the Application components (including all of the *Application*-dependent components) to the chosen device(s).

The *create* operation shall execute the Application components (including all of the *Application*-dependent components) using the entry points dictated in the SPD's implementation code element. Parameters passed to entry points will be (/ DomainName / NodeName / [other context sequences]) / ComponentName\_UniqueIdentifier . The unique identifier is determined by the implementation, unique to each node. The *create* operation uses this naming string to form component names that need to be retrieved from Naming Service. (See also section 3.2.1.3.) Due to the dynamics of bind and resolve to Naming Service, the *create* operation shall provide sufficient attempts to retrieve component object references from Naming Service prior to generating an exception.

The *create* shall initialize an *Application* component provided the component implements the CF *LifeCycle* interface.

The *create* shall configure an *Application* component provided the component implements the CF *PropertySet* interface. The *configure* operation input parameters can either be the configuration properties specified by the ApplicationProfile or the client (user) properties passed in the *create* call.

When client Application properties are supplied in the *create* call, the *ApplicationFactory* shall use the client supplied properties for the *configure* call. When the client does not supply properties, the *create* operation shall configure the component using properties supplied by the SCD if these properties have values associated with them.

The *create* operation shall interconnect *Application* components' (*Resources*' or *Devices*') ports in accordance with the SAD. It shall be possible to obtain *Ports* in accordance with the SAD via CF *Resource::getPort* operation. It shall be possible to obtain a CF *Resource* in accordance with the SAD via the CORBA Naming Service, *ResourceFactory*, or a stringified IOR. The *ResourceFactory* can be obtained by using the CORBA NamingService or a stringified IOR as stated in the SAD.

The *ApplicationFactory* shall pass, with invocation of the *ResourceFactory createResource* operation, all the *ResourceFactory* configuration properties as dictated by the SAD.

The dependencies to *Logger*, *File*, *Device*, and CORBA NamingService will show up as connections in the SAD.

If creation of components needs to be authenticated, then the *create* operation shall use an authentication service before performing any other operations.

If port connections between components need to be authenticated, then the *create* operation shall use an authentication service before connecting *Ports* together.

If port connections between components need to be access-controlled during execution, then the *create* operation shall update an access control database.

If the *Application* is successfully created, the *ApplicationFactory* shall return an *Application* component reference for the created *Application*. A sequence of created *Application* references can be obtained using the *DomainManager getApplications* operation.

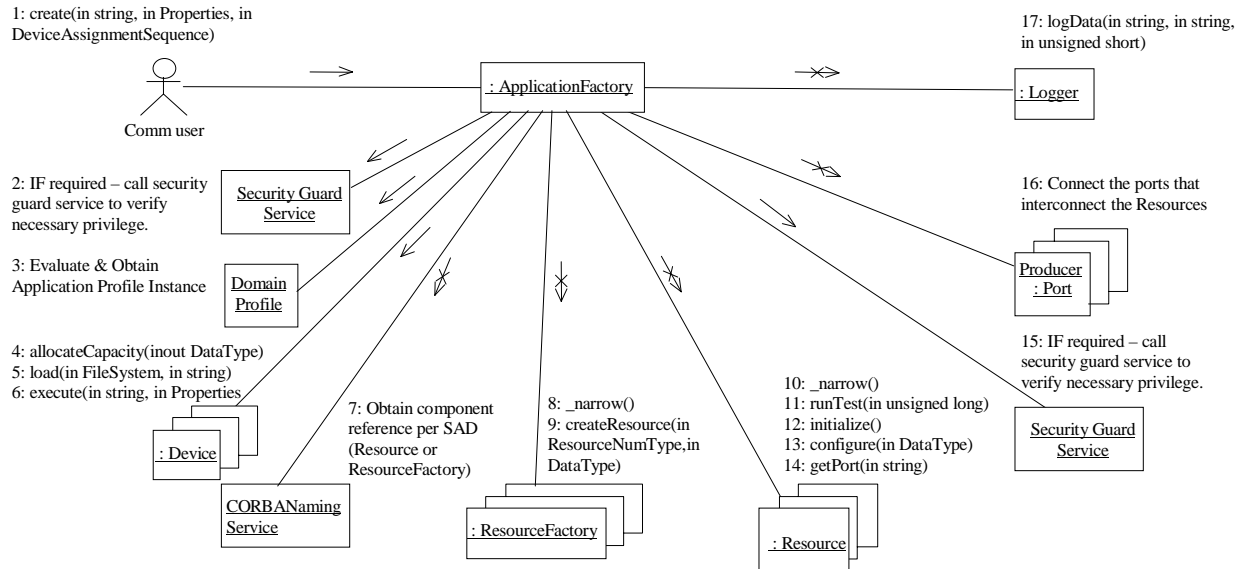
The *ApplicationFactory* shall, upon successful *Application* creation, log an Administrative event with the *DomainManager's Logger*.

The *ApplicationFactory* shall, upon unsuccessful *Application* creation, log an Alarm event with the *DomainManager's Logger*.

The following steps demonstrate one scenario of the *ApplicationFactory's* behavior for the creation of an *Application*:

1. Client invokes the *create* operation.
2. Call security guard service for verification of necessary privilege (if required).
3. Evaluate the Domain Profile for available *Devices* that meet the *Application's* memory and processor requirements, available Dependent Applications (e.g., *I/O* or *Utility* resources), and dependent libraries needed by the *Application*. Create an *ApplicationProfile* instance if the requested *Application* can be created. Update the *Device(s)* memory and processor utilization.
4. Allocate the *Device(s)* memory and processor utilization.
5. Load the *Application* components on the devices using the appropriate *Device(s)* interface provided the *Application* component hasn't already been loaded.
6. Execute the *Application* components on the devices using the appropriate *Device* interface as indicated by the *ApplicationProfile*.
7. Obtain the component reference (*Resource* or *ResourceFactory*) as described by the SAD.
8. If the component obtained from CORBA Naming Services is a *ResourceFactory* as indicated by the SAD, then narrow the component reference to be a *ResourceFactory* component.
9. If the component is an *ResourceFactory*, then create a *Resource* using the *ResourceFactory* interface.
10. If the components obtained from Naming Services are *Resources* supporting the *Resource* interface as indicated by the SCDs, then narrow the components reference to be *Resource* components.
11. If the *ApplicationProfile* dictates startup self tests the *ApplicationFactory* invokes the *runTest* operation of the *Application*.
12. Initialize the *Application*.
13. Configure the *Application*.
14. Get ports for the resources in order to interconnect the *Resources's* ports together.
15. Call security guard service for verification of necessary privilege (if required).
16. Connect the ports that interconnect the *Resources's* ports together.
17. Notify the client that the *Application* was instantiated successfully.

Figure 3-12 is a collaboration diagram depicting the behavior as described above.



**Figure 3-12. ApplicationFactory Behavior**

#### 3.1.3.2.2.5.1.4 Returns.

This operation returns a duplicated *Application* reference for the created *Application*.

#### 3.1.3.2.2.5.1.5 Exceptions/Errors.

The *create* operation shall raise the *CreateApplicationRequestError* exception when the parameter *DeviceAssignmentSequence* contains one (1) or more invalid *Application* component to device assignment.

The *create* operation shall raise the *CreateApplicationError* exception when the *create* request is valid but the *Application* can not be successfully instantiated due to internal processing error(s).

#### 3.1.3.2.3 DomainManager.

##### 3.1.3.2.3.1 Description.

The *DomainManager* interface is for the control and configuration of the system domain.

The *DomainManager* interface can be logically grouped into three categories: Human Computer Interface (HCI), Registration, and CF administration.

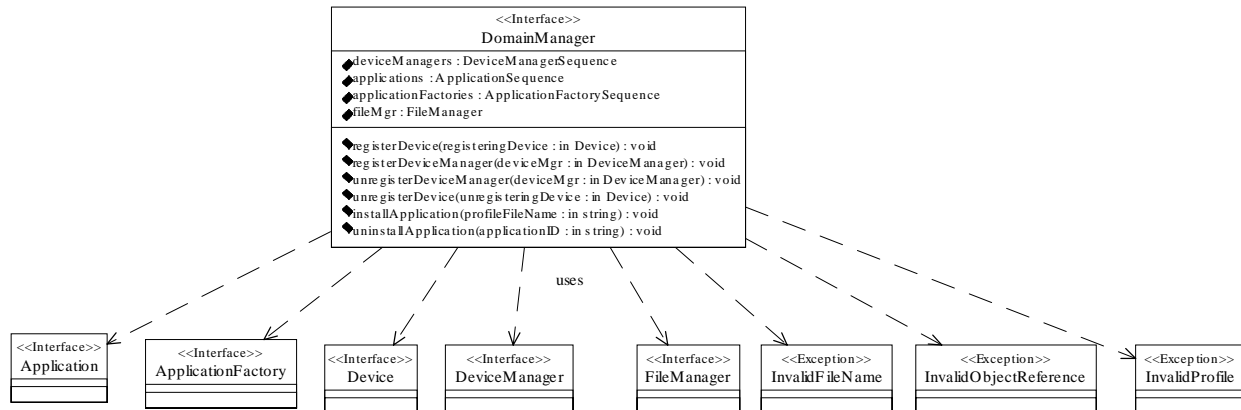
The HCI operations are used to configure the domain, get the domain's capabilities (*Devices* and *Applications*), and initiate maintenance functions. Host operations are performed by an HCI client capable of interfacing to the *DomainManager*.

The registration operations are used to register / unregister *DeviceManagers* and *DeviceManager's Devices* and *Applications* at startup or during run-time for dynamic device and Application extraction and insertion.

The administration operations are used to access the interfaces of registered *DeviceManagers* and *FileManagers*.



## 3.1.3.2.3.2 UML.

Figure 3-13. *DomainManager* Interface UML

## 3.1.3.2.3.3 Types.

## 3.1.3.2.3.3.1 exception ApplicationInstallationError.

This exception type is raised when an Application installation has not completed correctly.

```
exception ApplicationInstallationError {};
```

## 3.1.3.2.3.3.2 InvalidIdentifier.

This exception indicates an application identifier is invalid.

```
exception InvalidIdentifier {};
```

## 3.1.3.2.3.3.3 DeviceManagerSequence.

This type defines an unbounded sequence of CF *DeviceManager*(s).

```
Typedef sequence <CF::DeviceManager> DeviceManagerSequence
```

## 3.1.3.2.3.3.4 ApplicationSequence.

This type defines an unbounded sequence of CF *Application*(s).

```
Typedef sequence <CF::Application> ApplicationSequence
```

## 3.1.3.2.3.3.5 ApplicationFactorySequence.

This type defines an unbounded sequence of CF *ApplicationFactory*(s).

```
Typedef sequence <CF::ApplicationFactory> ApplicationFactorySequence
```

## 3.1.3.2.3.4 Attributes.

## 3.1.3.2.3.4.1 deviceManagers.

The deviceManagers attribute is read-only containing a sequence of registered *DeviceManagers* in the domain. If there are no registered *DeviceManagers*, the *DomainManager* shall return a *DeviceManagerSequence* with the sequence length set to 0. The *DomainManager* shall log an Administrative event with the *DomainManager's Logger* when the deviceManagers attribute is obtained.

```
readonly attribute DeviceManagerSequence      deviceManagers;
```

#### 3.1.3.2.3.4.2 applications.

The applications attribute is read-only containing a sequence of instantiated *Applications* in the domain. If there are no instantiated *Applications*, the *DomainManager* shall return an *ApplicationSequence* with the sequence length set to 0. The *DomainManager* shall log an Administrative event with the *DomainManager's Logger* when the applications attribute is obtained.

```
readonly attribute ApplicationSequence      applications;
```

#### 3.1.3.2.3.4.3 applicationFactories.

The applicationFactories attribute is read-only containing a sequence of *ApplicationFactories* in the domain. If there are no instantiated *ApplicationFactories* in the domain, the *DomainManager* shall return an *ApplicationFactorySequence* with the sequence length set to 0. The *DomainManager* shall log an Administrative event with the *DomainManager's Logger* when the applicationFactories attribute is obtained.

```
readonly attribute ApplicationFactorySequence
      applicationFactories;
```

#### 3.1.3.2.3.4.4 fileMgr.

The fileMgr attribute is read only containing the mounted FileSystems in the domain. The *DomainManager* shall log an Administrative event with the *DomainManager's Logger* when the fileMgr attribute is obtained.

```
readonly attribute FileManager      fileMgr;
```

#### 3.1.3.2.3.5 General Class Behavior.

During component construction the *DomainManager* shall register itself with the CORBA Naming Service if available. During Naming Service registration the *DomainManager* shall create a naming context of *"/Domain\_Name"* and bind the *DomainManager's* name with the created context. (If the CORBA Naming Service is not provided in an OE, *DomainManager* clients (e.g., *DeviceManager*) will have stringified IORs in their software profile entry in the Domain Profile.)

The *DomainManager* shall maintain a *Logger* component reference for logging.

The *DomainManager* shall create its own *FileManager* component that consists of all registered *DeviceManager's FileSystems*.

#### 3.1.3.2.3.6 Operations.

##### 3.1.3.2.3.6.1 registerDeviceManager.

##### 3.1.3.2.3.6.1.1 Brief Rationale.

This operation is used to register a *DeviceManager* and its *Device(s)*. Software profiles can also be obtained from the *DeviceManager's FileManager*.

##### 3.1.3.2.3.6.1.2 Synopsis.

```
void registerDeviceManager(in DeviceManager deviceMgr) raises (
CF::InvalidObjectReference );
```

### 3.1.3.2.3.6.1.3 Behavior.

The *registerDeviceManager* operation shall verify that the input parameter, *deviceMgr*, is a not a nil CORBA component reference.

The *registerDeviceManager* operation shall obtain the *DeviceManager Devices* using the CF *DeviceManager* interface.

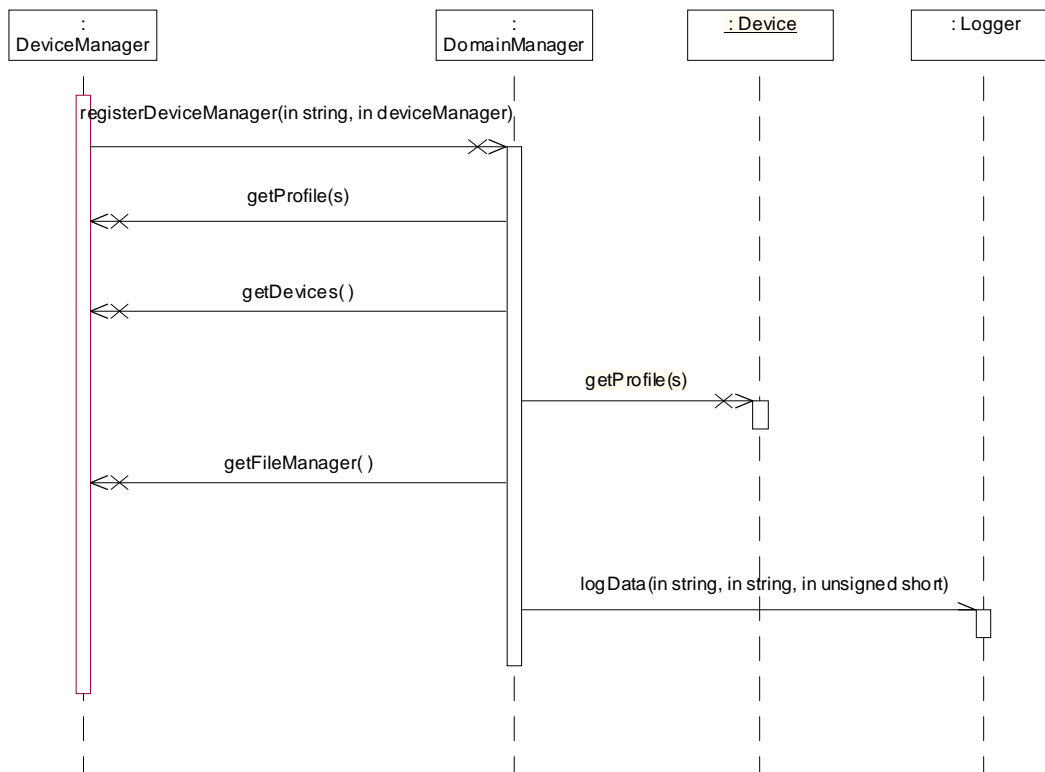
For each *Device* the *registerDeviceManager* operation shall obtain the attributes for the *Device* using the CF *Device* interface. The *Device* attributes contain the Device Profile and State information.

The *registerDeviceManager* operation shall obtain all the Software profiles from the registering *DeviceManager's FileSystems*.

The *registerDeviceManager* operation shall mount all *DeviceManager's FileSystems* to its *FileManager*. The mounted *FileSystem* names shall be unique and of the format: “/DomainName/HostName/FileSystemName”.

The *registerDeviceManager* operation shall, upon unsuccessful *DeviceManager* registration, log an Alarm event with the *DomainManager's Logger*.

The following UML sequence diagram (figure 3-14) illustrates the *DomainManager's* behavior for the *registerDeviceManager* operation.



**Figure 3-14. DomainManager Sequence Diagram for RegisterDeviceManager Operation**

#### 3.1.3.2.3.6.1.4 Returns.

This operation does not return a value.

#### 3.1.3.2.3.6.1.5 Exceptions/Errors.

The CF *InvalidObjectReference* exception shall be raised by the *registerDeviceManager* operation when the input parameter *DeviceManager* contains an invalid reference to a *DeviceManager* interface.

#### 3.1.3.2.3.6.2 registerDevice.

##### 3.1.3.2.3.6.2.1 Brief Rationale.

This operation is used to register a device for a specific *DeviceManager* in the *DomainManager's* Domain Profile.

The registering device could have been automatically detected by the *DeviceManager* or by the installation of a new device profile to the *DeviceManager's* file system by an installer application. For either case, the *DeviceManager* registers the *Device* with the *DomainManager*.

##### 3.1.3.2.3.6.2.2 Synopsis.

```
void registerDevice(in Device registeringDevice) raises (
CF::InvalidObjectReference );
```

##### 3.1.3.2.3.6.2.3 Behavior.

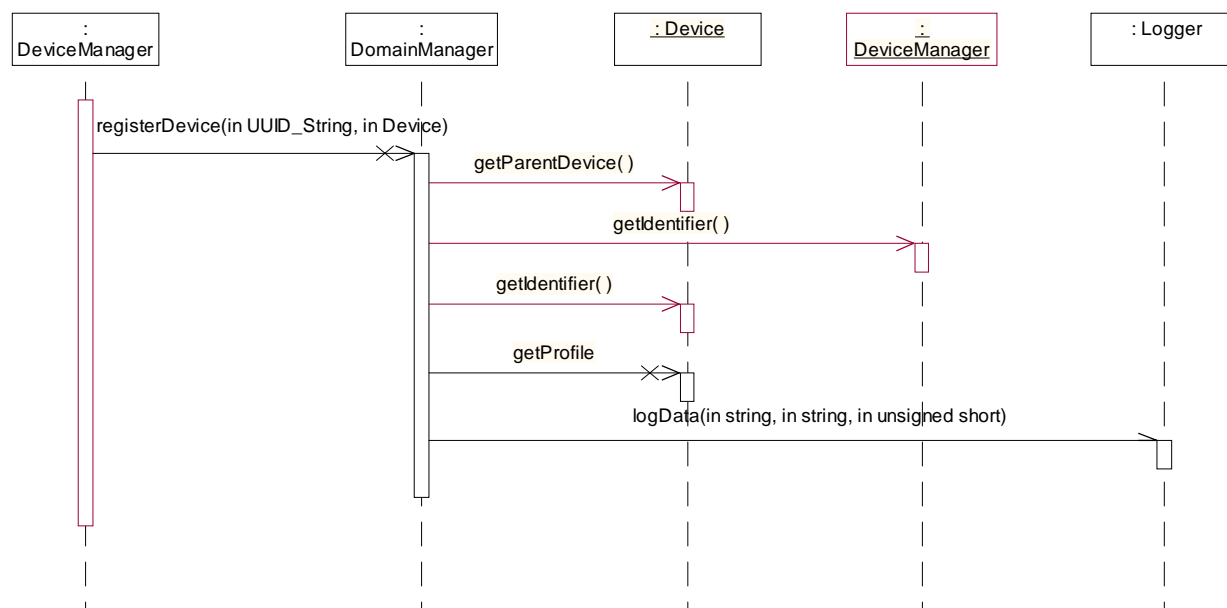
The *registerDevice* operation shall verify the input device is a not a nil CORBA component reference.

The *registerDevice* operation shall verify the *Device's* parent *DeviceManager* is a valid registered *DeviceManager*.

The *registerDevice* operation shall upon successful device registration, log an Administrative event with the *DomainManagers Logger*.

The *registerDevice* operation shall, upon unsuccessful device registration, log an Alarm event with the *DomainManager's Logger*.

The following UML sequence diagram (figure 3-15) illustrates the *DomainManager's* behavior for the *registerDevice* operation.



**Figure 3-15. *DomainManager* Sequence Diagram for *RegisterDevice* Operation**

#### 3.1.3.2.3.6.2.4 Returns.

This operation does not return a value.

#### 3.1.3.2.3.6.2.5 Exceptions/Errors.

The CF *InvalidObjectReference* exception shall be raised by the *registerDevice* operation when input parameter *registeringDevice* contains an invalid reference to a device interface.

#### 3.1.3.2.3.6.3 *installApplication*.

##### 3.1.3.2.3.6.3.1 Brief Rationale.

This operation is used to install new Application software in the *DomainManager's* Domain Profile.

An installer application typically invokes this operation when it has completed the installation of a new Application into the domain.

##### 3.1.3.2.3.6.3.2 Synopsis.

```
void installApplication(in string profileFileName) raises ( InvalidProfile,
CF::InvalidFileName, ApplicationInstallationError);
```

##### 3.1.3.2.3.6.3.3 Behavior.

The *profileFileName* is the absolute path of the Software Profile.

The *installApplication* operation shall verify the new Software Profile exists in the main *FileManager* repository and all the files the Application is dependent on are also resident.

The *registerApplication* operation shall add the Software Profile entry into the *DomainManager's* Domain Profile.

The *installApplication* operation shall, upon successful Application installation, create an *ApplicationFactory* and log an Administrative event with the *DomainManager's* *Logger*.

The *installApplication* operation shall, upon unsuccessful application installation, log an Alarm event with the *DomainManager's Logger*.

#### 3.1.3.2.3.6.3.4 Returns.

This operation does not return a value.

#### 3.1.3.2.3.6.3.5 Exceptions/Errors.

The CF InvalidProfile exception shall be raised by the *installApplication* operation when the input parameter *profileFileName* is invalid.

The CF InvalidFileName exception shall be raised by the *installApplication* operation when the input Software Profile file name is invalid.

The ApplicationInstallationError exception shall be raised by the *installApplication* operation when the installation of the Application file(s) was not successfully completed.

#### 3.1.3.2.3.6.4 *unregisterDeviceManager*.

##### 3.1.3.2.3.6.4.1 Brief Rationale.

This operation is used to unregister a *DeviceManager* component from the *DomainManager's* Domain Profile. A *DeviceManager* may be unregistered during run-time for dynamic extraction or maintenance of the *DeviceManager*.

##### 3.1.3.2.3.6.4.2 Synopsis.

```
void unregisterDeviceManager(in DeviceManager deviceMgr) raises (
CF::InvalidObjectReference);
```

##### 3.1.3.2.3.6.4.3 Behavior.

The *unregisterDeviceManager* operation shall unregister a *DeviceManager* component from the *DomainManager*.

The *unregisterDeviceManager* operation shall verify all *Devices* that are associated with the *DeviceManager* are in an Administrative Locked State prior to unregistering the *DeviceManager*.

The *unregisterDeviceManager* operation shall release all device(s) associated with the *DeviceManager* that is being unregistered.

The *unregisterDeviceManager* operation shall unmount all *DeviceManager's FileSystems* from its *File Manager*.

The *unregisterDeviceManager* operation shall, upon the successful unregistration of a *DeviceManager*, log an Administrative event with the *DomainManager's Logger*.

The *unregisterDeviceManager* operation shall, upon unsuccessful unregistration of a *DeviceManager*, log an Alarm event with the *DomainManager's Logger*.

#### 3.1.3.2.3.6.4.4 Returns.

This operation does not return a value.

#### 3.1.3.2.3.6.4.5 Exceptions/Errors.

The CF InvalidAdminState exception shall be raised by the *unregisterDeviceManager* operation when any of the *DeviceManager's Devices* are not in an Administrative Locked State.

The CF InvalidObjectReference exception shall be raised by the *unregisterDeviceManager* operation when the input parameter *DeviceManager* contains an invalid reference to a *DeviceManager* interface.

#### 3.1.3.2.3.6.5 *unregisterDevice*.

##### 3.1.3.2.3.6.5.1 Brief Rationale.

This operation is used to remove a device entry from the *DomainManager* for a specific *DeviceManager*.

##### 3.1.3.2.3.6.5.2 Synopsis.

```
void unregisterDevice(in Device unregisteringDevice) raises (
CF::InvalidObjectReference);
```

##### 3.1.3.2.3.6.5.3 Behavior.

The *unregisterDevice* operation shall remove a device entry from the *DomainManager's* Domain Profile for a registered *DeviceManager*.

The *unregisterDevice* operation shall verify the *Device* is in an Administrative Lock State prior to unregistering the Device.

The *unregisterDevice* operation shall release the *DomainManager*-maintained *Device* CORBA object reference.

The *unregisterDevice* operation shall, upon the successful unregistration of a *Device*, log an Administrative event with the *DomainManager's* Logger.

The *unregisterDevice* operation shall, upon unsuccessful unregistration of a *Device*, log an Alarm event with the *DomainManager's* Logger.

##### 3.1.3.2.3.6.5.4 Returns.

This operation does not return a value.

##### 3.1.3.2.3.6.5.5 Exceptions/Errors.

The CF InvalidAdminState exception shall be raised by the *unregisterDevice* operation when the *Device* is not in an Administrative Lock State.

The CF InvalidObjectReference exception shall be raised by the *unregisterDevice* operation when the input parameter contains an invalid reference to a Device interface.

#### 3.1.3.2.3.6.6 *uninstallApplication*.

##### 3.1.3.2.3.6.6.1 Brief Rationale.

This operation is used to uninstall an *ApplicationFactory* in the *DomainManager's* Domain Profile.

An installer application (section 3.2.2.1.1) typically invokes this operation when removing an *ApplicationFactory* from the domain.

##### 3.1.3.2.3.6.6.2 Synopsis.

```
void uninstallApplication(in String ApplicationID);
```

### 3.1.3.2.3.6.6.3 Behavior.

The *uninstallApplication* operation shall remove all files associated with the *ApplicationFactory* and Software Profile.

The *uninstallApplication* operation shall tear down the *ApplicationFactory*.

The *uninstallApplication* operation shall, upon successful uninstall of an *Application*, log an Administrative event with the *DomainManager's Logger*.

The *uninstallApplication* operation shall, upon unsuccessful uninstall of an *Application*, log an Alarm event with the *DomainManager's Logger*.

The *uninstallApplication* operation shall not allow any new *create* requests for this *ApplicationFactory*.

### 3.1.3.2.3.6.6.4 Returns.

This operation does not return a value.

### 3.1.3.2.3.6.6.5 Exceptions/Errors.

The *InvalidIdentifier* exception shall be raised when the *ApplicationID* is invalid.

### 3.1.3.2.4 Device.

#### 3.1.3.2.4.1 Description.

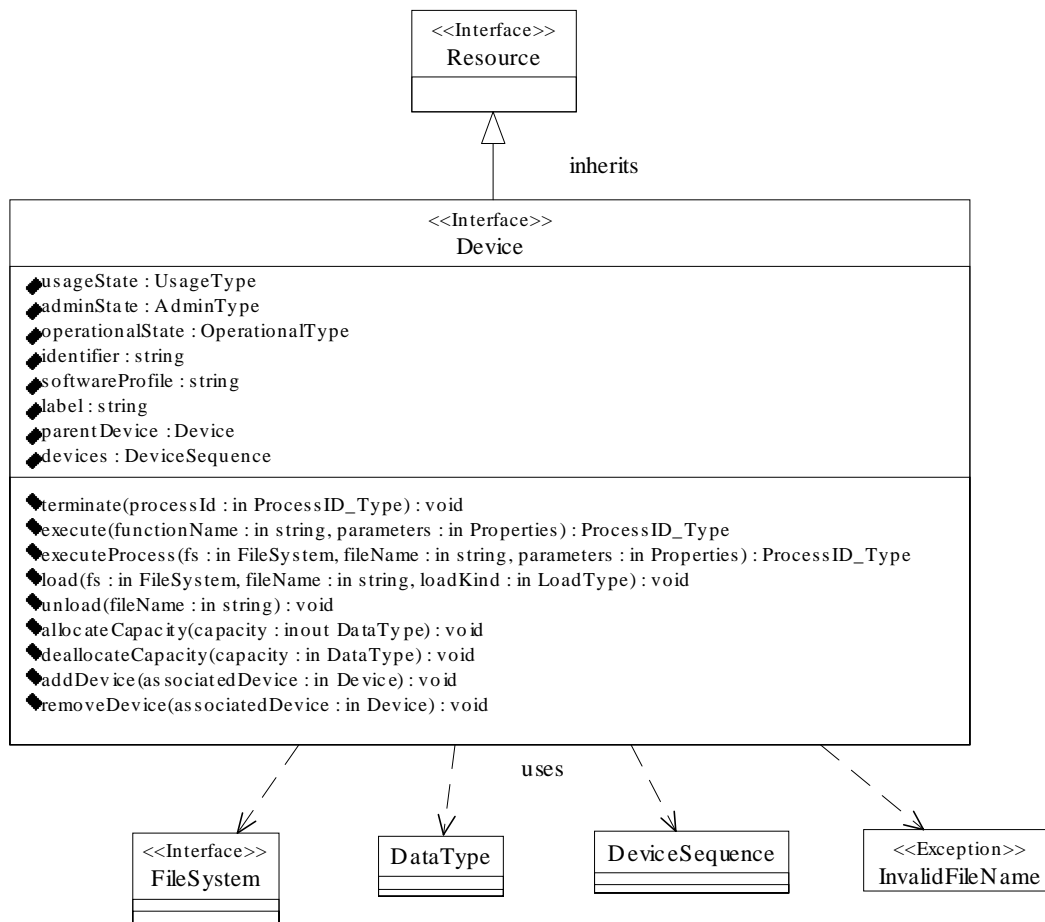
This interface defines the capabilities and attributes for any logical device in the domain. A logical device is an abstraction of a hardware device. There may be many logical devices per hardware device within the system. Alternately, the logical device may not necessarily represent actual hardware in the system. This is analogous to the loopback logical device in TCP/IP, which appears to the operating system as a network interface device, yet it's not a driver for hardware. The interface for a *Device* is based upon its properties, which are:

1. Device Profile – This XML profile defines the kind of hardware device and the applicable attributes for type of device. The Device Profile also indicates if the device has load and execute behavior and the attributes that can be queried and configured.
2. Software Profile – This XML profile defines the logical device capabilities (data ports, configure attributes, status attributes, etc.) which could be a subset of the hardware capabilities.
3. State Management & Status Attributes - This information is based upon the X.731 INFORMATION TECHNOLOGY - OPEN SYSTEMS INTERCONNECTION - SYSTEMS MANAGEMENT: STATE MANAGEMENT FUNCTION. The identical text is also published as ISO/IEC International Standard 10164-2.
4. Capacity Attributes – This information is the set of capacity attributes (e.g., memory, performance, etc.) for a device. In order to use a device certain capacities have to be obtained from the *Device*. The capacity attributes will vary among devices.

The *Device* is responsible for loading and executing software on a processor if it has loading and execution capabilities.



## 3.1.3.2.4.2 UML.

Figure 3-16. *Device Interface UML*

## 3.1.3.2.4.3 Types.

## 3.1.3.2.4.3.1 InvalidProcess Exception.

This exception indicates that a process with that ID does not exist on this device.

```
exception InvalidProcess {};
```

## 3.1.3.2.4.3.2 InvalidFunction Exception.

This exception indicates that a function with that name hasn't been loaded on this device.

```
exception InvalidFunction {};
```

## 3.1.3.2.4.3.3 DeviceNotCapable Exception.

This exception indicates that the device is not capable of the behavior being attempted such as *execute*.

```
exception DeviceNotCapable {};
```

#### 3.1.3.2.4.3.4 InvalidCapacity Exception.

This exception indicates that the capacity is unknown by this device or the capacity data value is invalid (wrong type).

```
exception Invalid Capacity {string msg;};
```

#### 3.1.3.2.4.3.5 CapacityExceeded Exception.

This exception indicates that the capacity allocation has been exceeded.

```
exception CapacityExceeded {};
```

#### 3.1.3.2.4.3.6 AdminType.

This is a CORBA IDL enumeration type that defines an object's administration states. The administration state indicates the permission to use or prohibition against using the *Resource*.

```
enum AdminType
{
    LOCKED,
    SHUTTING_DOWN,
    UNLOCKED
};
```

#### 3.1.3.2.4.3.7 OperationalType.

This is a CORBA IDL enumeration type that defines an object's Operational states. The Operational state indicates whether or not the object is working or not.

```
enum OperationalType
{
    ENABLED,
    DISABLED
};
```

#### 3.1.3.2.4.3.8 UsageType.

This is a CORBA IDL enumeration type that defines the object's Usage states. This state indicates whether or not an object is actively in use at a specific instant, and if so, whether or not it has spare capacity for additional users at that instant.

```
enum UsageType
{
    IDLE,
    ACTIVE,
    BUSY
};
```

#### 3.1.3.2.4.3.9 ProcessID\_Type.

This defines the process number within the system. Process number is unique to the Processor operating system that created the process.

```
typedef unsigned long ProcessID_Type;
```

#### 3.1.3.2.4.3.10 LoadType.

This type defines the type of load to be performed. The loading of the software can be performed as a driver, in the OS kernel memory, or as a relocatable object.

```
enum LoadType
{
    KERNEL_MODULE,
    RELOCATABLE_OBJECT,
    DRIVER
};
```

#### 3.1.3.2.4.4 Attributes.

##### 3.1.3.2.4.4.1 UsageState.

This state indicates whether or not a device is actively in use at a specific instant, and if so, whether or not it has spare capacity for additional users at that instant.

```
readonly attribute UsageType usageState;
```

##### 3.1.3.2.4.4.2 AdminState.

The administration state indicates the permission to use or prohibition against using the device. Some of the administration states are settable, but the shutting down state is not settable.

```
attribute AdminType adminState;
```

##### 3.1.3.2.4.4.3 OperationalState.

The operational state indicates whether or not the *Device* is working or not.

```
attribute OperationalType operationalState;
```

##### 3.1.3.2.4.4.4 Identifier.

The identifier attribute is the unique identifier for a device instance.

```
readonly attribute string identifier;
```

##### 3.1.3.2.4.4.5 SoftwareProfile.

The profile attribute is the XML software definition for this logical device.

```
readonly attribute string softwareProfile;
```

##### 3.1.3.2.4.4.6 Label.

The Label attribute is the label for this device. The attribute could convey location information within the system (e.g., audio1, serial1, etc.).

```
readonly attribute string label;
```

##### 3.1.3.2.4.4.7 ParentDevice.

The ParentDevice attribute indicates the parent device this device is associated with by either being a part of or was created from.

```
readonly attribute Device parentDevice;
```

##### 3.1.3.2.4.4.8 Devices.

The Devices attribute contains a list of devices associated with the *Device*. The association is an aggregate relationship. This aggregate relationship can be very tightly coupled or loosely coupled. Tightly coupled is a composition relationship where the device cannot exist without the

other associated device and cannot be associated with another device. Loosely coupled means the device can be associated with another device.

readonly attribute DeviceSequence devices;

### 3.1.3.2.4.5 Operations.

#### 3.1.3.2.4.5.1 *terminate*.

##### 3.1.3.2.4.5.1.1 Brief Rationale.

The *terminate* operation provides the mechanism for terminating the execution of an application on a specific device that was started up with the *execute* operation.

##### 3.1.3.2.4.5.1.2 Synopsis.

```
void terminate(in ProcessID_Type processId) raise ( InvalidProcess,
DeviceNotCapable );
```

##### 3.1.3.2.4.5.1.3 Behavior.

The *terminate* operation shall terminate the execution of the function on the device.

##### 3.1.3.2.4.5.1.4 Returns.

This operation does not return a value.

##### 3.1.3.2.4.5.1.5 Exceptions/Errors.

The InvalidProcess exception shall be raised when the processID does not exist for that device.

The DeviceNotCapable exception shall be raised when the *Device* is not capable of this behavior.

#### 3.1.3.2.4.5.2 *execute*.

##### 3.1.3.2.4.5.2.1 Brief Rationale.

This operation provides the mechanism for starting up and executing software remotely on a device, providing the device is capable of executing software threads/processes.

##### 3.1.3.2.4.5.2.2 Synopsis.

```
ProcessID_Type execute(in string functionName, in Properties parameters)
raises ( DeviceNotCapable, InvalidFunction );
```

##### 3.1.3.2.4.5.2.3 Behavior.

The *execute* operation shall execute the given function with the input parameters. The parameters (IDs and format values) shall be:

1. prefix naming context – The ID is 1 and the value is a CORBA string.
2. stringified IOR – The ID is 2 and the value is a CORBA string.

##### 3.1.3.2.4.5.2.4 Returns.

The *execute* operation shall return the ID of the process that has been created.

##### 3.1.3.2.4.5.2.5 Exceptions/Errors.

The DeviceNotCapable exception shall be raised when the device is not capable of executing software.

The InvalidFunction exception shall be raised when the function does not exist which means it hasn't been loaded on that device.

### 3.1.3.2.4.5.3 *load*.

#### 3.1.3.2.4.5.3.1 Brief Rationale.

This operation provides the mechanism for loading software on a specific device, thus allowing new software to be executed on that device.

#### 3.1.3.2.4.5.3.2 Synopsis.

```
void load(in FileSystem fs, in string fileName, in LoadType loadKind)
    raises ( DeviceNotCapable, InvalidFileName );
```

#### 3.1.3.2.4.5.3.3 Behavior.

The *load* operation shall load a file on the specified device based on the given loadKind and fileName using the input *FileSystem* to retrieve it. If the input *FileSystem* is nil, then the *load* operation shall use the parent *Device's FileManager* for finding the file to loaded.

#### 3.1.3.2.4.5.3.4 Returns.

This operation does not return any value.

#### 3.1.3.2.4.5.3.5 Exceptions/Errors.

The DeviceNotCapable exception shall be raised when the device is not capable of loading the application (e.g., serial, audio, Ethernet, etc.).

The CF InvalidFileName exception shall be raised when the file does not exist.

### 3.1.3.2.4.5.4 *unload*.

#### 3.1.3.2.4.5.4.1 Brief Rationale.

This operation provides the mechanism to unload software that was previously loaded.

#### 3.1.3.2.4.5.4.2 Synopsis.

```
void unload(in string fileName) raises ( DeviceNotCapable, InvalidFileName );
```

#### 3.1.3.2.4.5.4.3 Behavior.

The *unload* operation shall unload application software on the specified device based on the input fileName.

#### 3.1.3.2.4.5.4.4 Returns.

This operation does not return a value.

#### 3.1.3.2.4.5.4.5 Exceptions/Errors.

The DeviceNotCapable exception shall be raised when the device is not capable of unloading software (e.g., serial, audio, Ethernet, etc.).

The CF InvalidFileName exception shall be raised when the file does not exist.

### 3.1.3.2.4.5.5 *allocateCapacity*.

#### 3.1.3.2.4.5.5.1 Brief Rationale.

This operation provides the mechanism to request and allocate additional capacity from the device, in order to perform more functions on the device or to use the device.

#### 3.1.3.2.4.5.5.2 Synopsis.

```
void allocateCapacity(inout DataType capacity) raises ( InvalidCapacity,
    CapacityExceeded );
```

### 3.1.3.2.4.5.5.3 Behavior.

This operation requests capacity from the device. The current capacity of the device shall be reduced according to the capacity model based upon the capacity requested.

### 3.1.3.2.4.5.5.4 Returns.

The *allocateCapacity* operation shall return the remaining available capacity of the device.

### 3.1.3.2.4.5.5.5 Exceptions/Errors.

The InvalidCapacity exception shall be raised when the capacity is invalid or the capacity value is the wrong type. The InvalidCapacity exception will state the reason for the exception.

The CapacityExceeded exception shall be raised when the capacity requested exceeds the allocable capacity of the device

### 3.1.3.2.4.5.6 *deallocateCapacity*.

#### 3.1.3.2.4.5.6.1 Brief Rationale.

This operation provides the mechanism to return capacity back to the device, in order to make the device available for other uses.

#### 3.1.3.2.4.5.6.2 Synopsis.

```
void deallocateCapacity(in DataType capacity) raises ( InvalidCapacity );
```

#### 3.1.3.2.4.5.6.3 Behavior.

This operation returns capacity to the device. The current capacity of the device shall be adjusted based upon the input capacity and the capacity model of the device.

#### 3.1.3.2.4.5.6.4 Returns.

This operation does not return any value.

#### 3.1.3.2.4.5.6.5 Exceptions/Errors.

The InvalidCapacity exception shall be raised when the capacity is invalid or the capacity value is the wrong type. The InvalidCapacity exception will state the reason for the exception.

### 3.1.3.2.4.5.7 *addDevice*.

#### 3.1.3.2.4.5.7.1 Brief Rationale.

This operation provides the mechanism to associate a *Device* with another *Device*. *Devices* within the system may be associated with one another. This operation provides the mechanism for creating this association. When a *Device* changes state or is being torn down, this affects its associated *Devices*. The *Device* being added is an aggregate piece of this *Device*. This aggregate relationship can be very tightly coupled or loosely coupled.

#### 3.1.3.2.4.5.7.2 Synopsis.

```
void addDevice(in Device associatedDevice);
```

#### 3.1.3.2.4.5.7.3 Behavior.

The *addDevice* operation shall add the new *Device* to its *Device's* attribute.

#### 3.1.3.2.4.5.7.4 Returns.

This operation does not return any value.

#### 3.1.3.2.4.5.7.5 Exceptions/Errors.

This operation does not raise any exceptions.

#### 3.1.3.2.4.5.8 removeDevice.

##### 3.1.3.2.4.5.8.1 Brief Rationale.

This operation provides the mechanism to disassociate a *Device* with another *Device*.

##### 3.1.3.2.4.5.8.2 Synopsis.

```
void removeDevice(in Device associatedDevice);
```

##### 3.1.3.2.4.5.8.3 Behavior.

The *removeDevice* operation shall remove the *Device* from its *Device's* attribute.

##### 3.1.3.2.4.5.8.4 Returns.

This operation does not return any value.

##### 3.1.3.2.4.5.8.5 Exceptions/Errors.

This operation does not raise any exceptions.

#### 3.1.3.2.4.5.9 executeProcess.

##### 3.1.3.2.4.5.9.1 Brief Rationale.

This operation provides the mechanism for starting up and executing software remotely on a device, providing the device is capable of executing software processes.

##### 3.1.3.2.4.5.9.2 Synopsis.

```
ProcessID_Type executeProcess(in FileSystem fs, in string fileName, in
Properties parameters)
    raises ( DeviceNotCapable, InvalidFileName );
```

##### 3.1.3.2.4.5.9.3 Behavior.

The *executeProcess* operation shall execute the given *fileName* with the input parameters. If the input *FileSystem* is nil, then the *executeProcess* operation shall use the parent *Device's FileManager* for finding the file to be executed. The valid IDs and format values for parameters shall be:

1. prefix naming context – The ID is 1 and the value is a CORBA string.
2. stringified IOR – The ID is 2 and the value is CORBA string.

##### 3.1.3.2.4.5.9.4 Returns.

The *executeProcess* operation shall return the ID of the process that has been created.

##### 3.1.3.2.4.5.9.5 Exceptions/Errors.

The *DeviceNotCapable* exception shall be raised when the device is not capable of executing software.

The *InvalidFileName* exception shall be raised when the *fileName* does not exist.

#### 3.1.3.2.4.5.10 Resource Operations.

A *Device* is a type of resource within the domain and has the requirements as stated in the *Resource* interface. In addition, the *releaseObject* operation shall cause the *Device* and its

associated *Devices* to be torn down. The *releaseObject* operation shall cause a *Device* administration state to transition to the shutting down state. When the *Device* administrative state is locked, meaning its associated *Devices* have been removed and usage state is idle, then the *Device* shall be torn down and released from the CORBA environment and removed from its parent device.

#### 3.1.3.2.5 *DeviceManager*.

##### 3.1.3.2.5.1 Description.

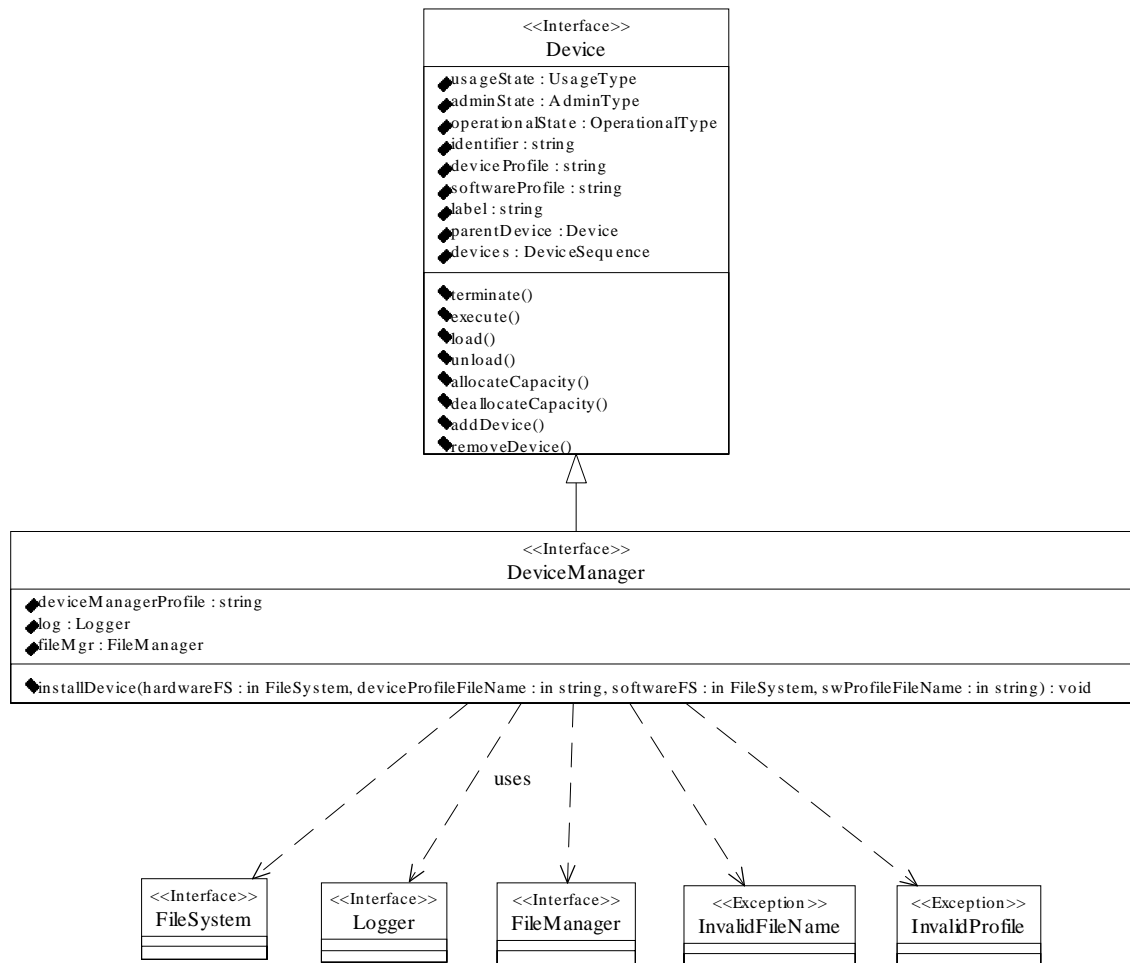
Each CORBA capable processor will have at least one *DeviceManager*. The interface for a *DeviceManager* is based upon its properties, which are:

1. *DeviceManager* Profile – This profile contains a mapping of physical device locations to meaning labels (e.g., audio1, serial1, etc.), along with its label-name and *DomainManager* context information. The *DeviceManager* profile allows for the same Device Profile to be referenced many times with the domain, instead of creating a device profile for each instance within the domain.
2. System Services - *Logger* and *FileManager*.

The *DeviceManager* interface extends the *Device* interface by adding device management operations for installing devices.



### 3.1.3.2.5.2 UML.



**Figure 3-17. DeviceManager UML**

### 3.1.3.2.5.3 Types.

N/A.

### 3.1.3.2.5.4 Attributes.

#### 3.1.3.2.5.4.1 Log.

The Log attribute is the CF *Logger* associated with this device manager.

readonly attribute `Logger log`;

#### 3.1.3.2.5.4.2 FileMgr.

The FileMgr attribute is the CF *FileManager* component associated with this device.

readonly attribute `FileManager fileMgr`;

### 3.1.3.2.5.4.3 DeviceManagerProfile.

The DeviceManagerProfile attribute contains the *DeviceManager*'s profile.

```
readonly attribute string deviceManagerProfile;
```

### 3.1.3.2.5.5 General Behavior.

The *DeviceManager* upon start up shall register itself with a *DomainManager*. This requirement allows the system to be developed where at a minimum only one component reference needs to be known which is the *DomainManager*. A *DeviceManager* shall use its profile for determining:

1. How to obtain the *DomainManager* component reference, whether NamingService is being used or a *DomainManager* stringified IOR is being used. When NamingService is used the *DeviceManager* shall create a Naming Context that uniquely identifies the *DeviceManager* node. The Naming Context shall be placed under the “/DomainName” Naming Context as “/DomainName/NodeName”.
2. Whether to create a *Logger* component or where to obtain a *Logger* component from.
3. The physical location for each *Device* Component.

The *DeviceManager* shall create *FileSystem* components implementing the CF *FileSystem* interface for each memory device (fixed or removable disks) resident on or being controlled by the *DeviceManager*. The *FileSystems* created shall be mounted to a *FileManager* component. Each mounted *FileSystem* name shall be unique within the domain. A *FileSystem* name of the format “/HostName/FileSystemName” shall be used.

### 3.1.3.2.5.6 Operations.

#### 3.1.3.2.5.6.1 installDevice.

##### 3.1.3.2.5.6.1.1 Brief Rationale.

This operation makes the *DeviceManager* aware that a new device is available for usage and where the hardware device and software logical Device Profiles are for this device. Depending on the implementation of the *DeviceManager*, it may or may not be aware (unable to detect) of a new device or the type of new device that it has found. The outcome of this operation is to create a new logical *Device*.

##### 3.1.3.2.5.6.1.2 Synopsis.

```
void installDevice(in FileSystem hardwareFS, in string deviceProfileFileName,
in FileSystem softwareFS, in string swProfileFileName)
    raises( InvalidFileName, InvalidProfile );
```

##### 3.1.3.2.5.6.1.3 Behavior.

The *installDevice* operation shall process the Device and Software Profiles, create a new *Device* component, and register *Device* with the *DomainManager*. Each *Device* created shall be given its logical device profile, its hardware device profile, identifier, label and parent device. If the *Device* is created up as a separate process, the *DeviceManager* object reference shall be passed as a parameter to the OS call that is used to create the logical *Device* process.

##### 3.1.3.2.5.6.1.4 Returns.

This operation does not return a value.

#### 3.1.3.2.5.6.1.5 Exceptions/Errors.

The CF InvalidFileName exception shall be raised when the file name does not exist in the input file system.

The CF InvalidProfile exception shall be raised when a profile is invalid.

#### 3.1.3.2.5.6.2 *addDevice*.

##### 3.1.3.2.5.6.2.1 Brief Rationale.

This operation provides the mechanism to associate a *Device* with another *Device*. *Devices* within the system may be associated with one another. This operation provides the mechanism for creating this association. When a *Device* changes state or is being torn down, this affects its associated *Devices*. The *Device* being added is an aggregate piece of this *Device*. This aggregate relationship can be very tightly coupled or loosely coupled.

##### 3.1.3.2.5.6.2.2 Synopsis.

```
void addDevice(in Device associatedDevice);
```

##### 3.1.3.2.5.6.2.3 Behavior.

The *addDevice* operation shall add the new *Device* to its *Device's* attribute. The *addDevice* operation shall register the device with the *DomainManager*. The registration shall be performed by either the *registerDeviceManager* process during power-up or by the installation of the device after *DeviceManager* registration with *the DomainManager*.

##### 3.1.3.2.5.6.2.4 Returns.

This operation does not return any value.

##### 3.1.3.2.5.6.2.5 Exceptions/Errors.

This operation does not raise any exceptions.

#### 3.1.3.2.5.6.3 *removeDevice*.

##### 3.1.3.2.5.6.3.1 Brief Rationale.

This operation provides the mechanism to disassociate a *Device* from another *Device*.

##### 3.1.3.2.5.6.3.2 Synopsis.

```
void removeDevice(in Device associatedDevice);
```

##### 3.1.3.2.5.6.3.3 Behavior.

The *removeDevice* operation shall remove the device from its *Device's* attribute. The *removeDevice* operation shall unregister the device with the *DomainManager*. The unregistration shall be performed by either the *unregisterDeviceManager* process during *releaseObject* of a *DeviceManager* or by the uninstalling (*releaseObject*) of the device after *DeviceManager* registration with *the DomainManager*.

##### 3.1.3.2.5.6.3.4 Returns.

This operation does not return any value.

##### 3.1.3.2.5.6.3.5 Exceptions/Errors.

This operation does not raise any exceptions.

#### 3.1.3.2.5.6.4 Resource Operations.

A *DeviceManager* is a type of device within the domain and has the requirements as stated in the *Device* interface. In addition, the *start* and *stop* shall have no effect on the *DeviceManager*. The *releaseObject* operation shall cause the *DeviceManager* and its associated *Devices* to be torn down (*releaseObject*). The *releaseObject* operation shall cause a *DeviceManager* administration state to transition to shutting down. When the *DeviceManager* admin state is locked, meaning its associated *Devices* removed and usage state is idle, then the *DeviceManager* shall be torn down and released from the CORBA environment, and unregistered with the *DomainManager*.

#### 3.1.3.3 Framework Services Interfaces.

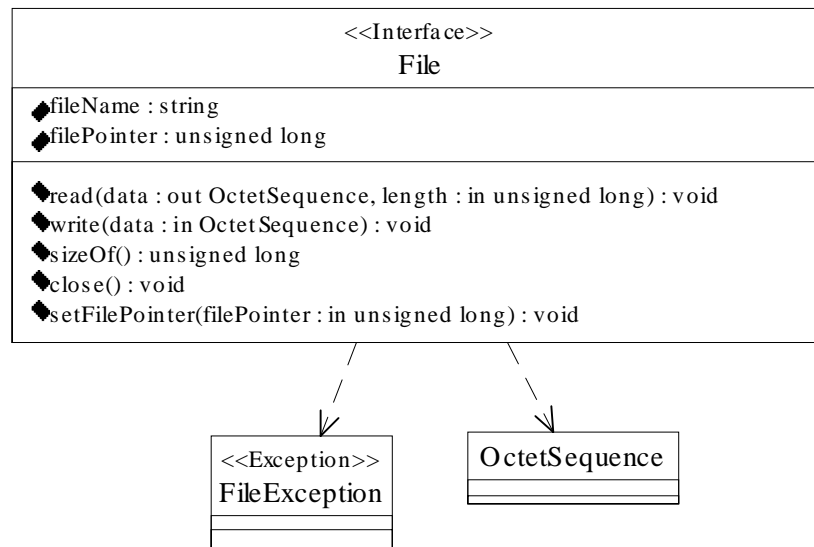
Framework Services Interfaces shall be implemented using the CF IDL presented in Appendix C.

##### 3.1.3.3.1 *File*.

##### 3.1.3.3.1.1 Description.

The *File* interface provides the ability to read and write files residing within a CF-compliant, distributed *FileSystem*. A file can be thought of conceptually as a sequence of octets with a current filepointer describing where the next read or write will occur. This filepointer points to the beginning of the file upon construction of the file object. The *File* interface is modeled after the POSIX/C file interface.

##### 3.1.3.3.1.2 UML.



**Figure 3-18. *File* Interface UML**

##### 3.1.3.3.1.3 Types.

##### 3.1.3.3.1.3.1 IOException.

```
exception IOException { string message; unsigned short errorCode};
```

This exception indicates an error occurred during a *read* or *write* operation to a *File*. The message shall provide additional information describing the reason why the error occurred.

### 3.1.3.3.1.3.2 InvalidFilePointer.

```
exception InvalidFilePointer {};
```

This exception indicates the file pointer is out of range based upon the current file size.

### 3.1.3.3.1.4 Attributes.

#### 3.1.3.3.1.4.1 FileName.

The `fileName` attribute provides read-only access to the fully qualified path of the file. The syntax for a filename is based upon the UNIX operating system. That is, a sequence of directory names separated by forward slashes (/) followed by the base filename. The `fileName` attribute will contain the filename given to the *FileSystem open* operation.

```
readonly attribute string fileName;
```

#### 3.1.3.3.1.4.2 FilePointer.

The `FilePointer` attribute provides read access to the file pointer position where the next read or write will occur.

```
Readonly attribute long filePointer;
```

### 3.1.3.3.1.5 Operations.

#### 3.1.3.3.1.5.1 *read*.

##### 3.1.3.3.1.5.1.1 Brief Rationale.

Applications require the *read* operation in order to retrieve data from remote files.

##### 3.1.3.3.1.5.1.2 Synopsis.

```
void read(out OctetSequence data, in unsigned long length) raises (
IOException);
```

##### 3.1.3.3.1.5.1.3 Behavior.

The *read* operation shall read octets from the file referenced up to the number specified by the length parameter and move the file pointer forward by the number of octets actually read. Less than this maximum number of octets shall be read only when an end of file is encountered.

##### 3.1.3.3.1.5.1.4 Returns.

The *read* operation shall return the number of octets actually read from the *File*. If the file pointer is pointing to the end of the *File*, the *read* operation shall return 0.

##### 3.1.3.3.1.5.1.5 Exceptions/Errors.

The `IOException` shall be raised when a *read* error occurs.

#### 3.1.3.3.1.5.2 *write*.

##### 3.1.3.3.1.5.2.1 Brief Rationale.

Applications require the *write* operation in order to write data to remote files.

##### 3.1.3.3.1.5.2.2 Synopsis.

```
void write(in OctetSequence data) raises ( IOException);
```

### 3.1.3.3.1.5.2.3 Behavior.

The *write* operation shall write the number of octets specified to the file referenced and move the file pointer forward by the number of octets written. The *write* operation shall write no data if an error occurs.

### 3.1.3.3.1.5.2.4 Returns.

None.

### 3.1.3.3.1.5.2.5 Exceptions/Errors.

The *IOException* shall be raised when a *write* error occurs. The file pointer shall remain unchanged if this exception is raised.

### 3.1.3.3.1.5.3 *sizeOf*.

#### 3.1.3.3.1.5.3.1 Brief Rationale.

An application may need to know the size of a file in order to determine memory allocation requirements.

#### 3.1.3.3.1.5.3.2 Synopsis.

```
unsigned long sizeOf() raises ( FileNotFoundException );
```

#### 3.1.3.3.1.5.3.3 Behavior.

There is no significant behavior beyond the behavior described by the following section.

#### 3.1.3.3.1.5.3.4 Returns.

The *sizeOf* operation shall return the number of octets stored in the file.

#### 3.1.3.3.1.5.3.5 Exceptions/Errors.

The CF *FileNotFoundException* shall be raised when a file-related error occurs (e.g., file does not exist anymore).

### 3.1.3.3.1.5.4 *close*.

#### 3.1.3.3.1.5.4.1 Brief Rationale.

The *close* operation is needed in order to release file resources once they are no longer needed.

#### 3.1.3.3.1.5.4.2 Synopsis.

```
void close() raises ( FileNotFoundException );
```

#### 3.1.3.3.1.5.4.3 Behavior.

The *close* operation shall release any OE file resources associated with the component. A closed file shall no longer be capable of *File*-related operations.

#### 3.1.3.3.1.5.4.4 Returns.

N/A.

#### 3.1.3.3.1.5.4.5 Exceptions/Errors.

The CF *FileNotFoundException* shall be raised when it can not successfully close the file.

### 3.1.3.3.1.5.5 *setFilePointer*.

#### 3.1.3.3.1.5.5.1 Brief Rationale.

The *setFilePointer* operation is needed in order position the file pointer where the next read or write will occur.

#### 3.1.3.3.1.5.5.2 Synopsis.

```
void setFilePointer(in unsigned long filePointer) raises (
InvalidFilePointer, FileException );
```

#### 3.1.3.3.1.5.5.3 Behavior.

The *setFilePointer* operation shall set the file pointer position to the input filePointer value.

#### 3.1.3.3.1.5.5.4 Returns.

This operation returns no value.

#### 3.1.3.3.1.5.5.5 Exceptions/Errors.

The CF FileException shall be raised when the file can not be successfully accessed to set the file pointer position.

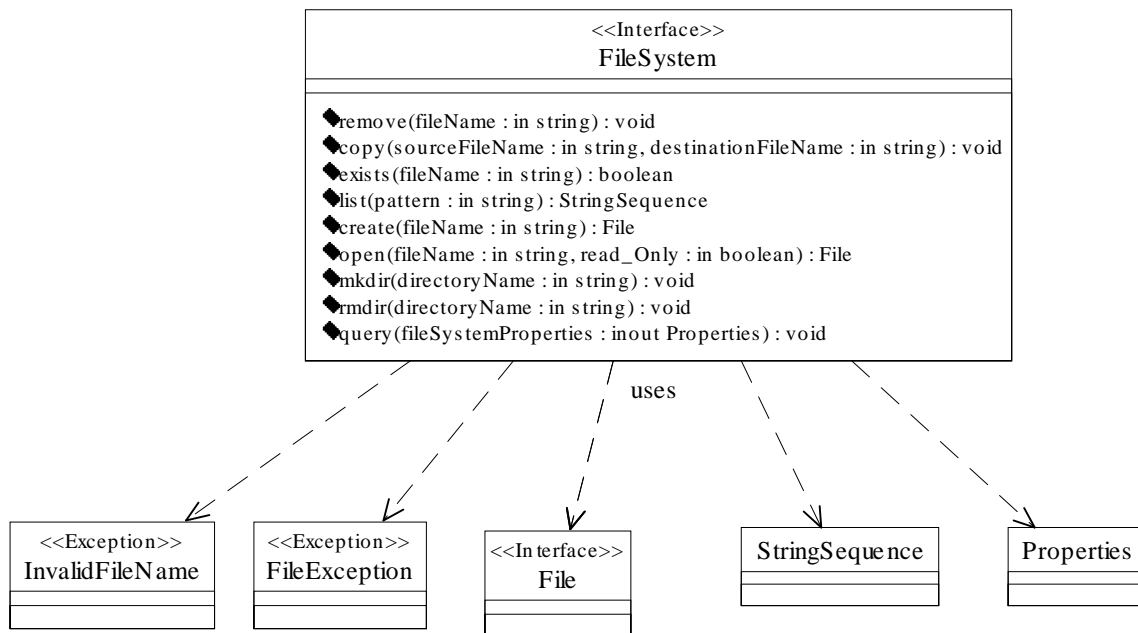
The InvalidFilePointer exception shall be raised when the file pointer exceeds the file size.

### 3.1.3.3.2 *FileSystem*.

#### 3.1.3.3.2.1 Description.

The *FileSystem* interface defines CORBA operations that enable remote access to a physical file system.

#### 3.1.3.3.2.2 UML.



**Figure 3-19. *FileSystem* Interface UML**

### 3.1.3.3.2.3 Types.

#### 3.1.3.3.2.3.1 UnknownFileSystemProperties.

```
exception UnknownFileSystemProperties {properties invalidProperties; };
```

This exception indicates a set of properties unknown by the component.

### 3.1.3.3.2.4 Attributes.

N/A.

### 3.1.3.3.2.5 Operations.

#### 3.1.3.3.2.5.1 *remove*.

##### 3.1.3.3.2.5.1.1 Brief Rationale.

The *remove* operation provides the ability to remove a file from a file system.

##### 3.1.3.3.2.5.1.2 Synopsis.

```
void remove(in string fileName) raises( FileException, InvalidFileName );
```

##### 3.1.3.3.2.5.1.3 Behavior.

The *remove* operation shall remove the file with the given filename. The *remove* operation shall ensure that the filename is an absolute pathname of the file relative to the target *FileSystem*.

##### 3.1.3.3.2.5.1.4 Returns.

None.

##### 3.1.3.3.2.5.1.5 Exceptions/Errors.

The InvalidFilename exception shall be raised when the filename is not valid.

The FileException shall be raised when a file-related error occurred during the *remove* operation.

#### 3.1.3.3.2.5.2 *copy*.

##### 3.1.3.3.2.5.2.1 Brief Rationale.

The *copy* operation provides the ability to copy a file to another file.

##### 3.1.3.3.2.5.2.2 Synopsis.

```
void copy(in string sourceFileName, in string destinationFileName) raises(
InvalidFileName, FileException );
```

##### 3.1.3.3.2.5.2.3 Behavior.

The *copy* operation shall copy the source file with the specified sourceFileName to the destination file with the specified destinationFileName. The *copy* operation shall ensure that the sourceFileName and destinationFileName are absolute pathnames relative to the target *FileSystem*.

##### 3.1.3.3.2.5.2.4 Returns.

None.

##### 3.1.3.3.2.5.2.5 Exceptions/Errors.

The InvalidFilename exception shall be raised when the filename is not valid.

The FileException shall be raised when a file-related error occurred during the *copy* operation.



### 3.1.3.3.2.5.3 *exists*.

#### 3.1.3.3.2.5.3.1 Brief Rationale.

The *exists* operation provides the ability to verify the existence of a file within a *FileSystem*.

#### 3.1.3.3.2.5.3.2 Synopsis.

```
boolean exists(in string fileName) raises( InvalidFileName );
```

#### 3.1.3.3.2.5.3.3 Behavior.

The *exists* operation shall check to see if a file exists based on the filename parameter. The *exists* operation shall ensure that the filename is a full pathname of the file relative to the target *FileSystem* and raise an exception if the name is invalid.

#### 3.1.3.3.2.5.3.4 Returns.

The *exists* operation shall return True if the file exists, or False if it does not.

#### 3.1.3.3.2.5.3.5 Exceptions/Errors.

The InvalidFilename exception shall be raised when filename is not valid.

### 3.1.3.3.2.5.4 *list*.

#### 3.1.3.3.2.5.4.1 Brief Rationale.

The *list* operation provides the ability to obtain a list of files in the *FileSystem* according to a given search pattern.

#### 3.1.3.3.2.5.4.2 Synopsis.

```
StringSequence list(in string pattern);
```

#### 3.1.3.3.2.5.4.3 Behavior.

The *list* operation shall return a list of filenames based upon the search pattern given. The following wildcard characters shall be supported:

- \* used to match any sequence of characters (including null).
- ? used to match any single character.

These wildcards may only be applied to the base filename in the search pattern given. For example, the following are valid search patterns:

- /tmp/files/\*.\* Returns all files and directories within the /tmp/files directory. Directory names shall be indicated with a "/" at the end of the name.
- /tmp/files/foo\* Returns all files beginning with the letters "foo" in the /tmp/files directory.
- /tmp/files/f?? Returns all 3 letter files beginning with the letter f in the /tmp/files directory.
- \*/files/foo\* Returns all files in subdirectories of the name "files" and starting with the letters "foo".
- \*/fi?es/f??.\* Returns all files in subdirectories of the name "fi", some character and "es" and starting with the letter "f" followed by any 2 characters.

#### 3.1.3.3.2.5.4.4 Returns.

The *list* operation shall return a StringSequence of filenames matching the wildcard specification.

## 3.1.3.3.2.5.4.5 Exceptions/Errors.

None.

3.1.3.3.2.5.5 *create*.

## 3.1.3.3.2.5.5.1 Brief Rationale.

The *create* operation provides the ability to create a new file on the *FileSystem*.

## 3.1.3.3.2.5.5.2 Synopsis.

```
File create(in string fileName) raises( InvalidFileName, FileException );
```

## 3.1.3.3.2.5.5.3 Behavior.

The *create* operation shall create a new *File* based upon the provided file name.

## 3.1.3.3.2.5.5.4 Returns.

The *create* operation returns a *File* component reference to the opened file. A null file component reference shall be returned if an error occurs.

## 3.1.3.3.2.5.5.5 Exceptions/Errors.

The InvalidFilename exception shall be raised when a filename is not valid.

The FileException shall be raised if the *File* already exists or another file error occurred.

3.1.3.3.2.5.6 *open*.

## 3.1.3.3.2.5.6.1 Brief Rationale.

The *open* operation provides the ability to open a file for read or write.

## 3.1.3.3.2.5.6.2 Synopsis.

```
File open(in string fileName, in boolean readOnly) raises( InvalidFileName, FileException );
```

## 3.1.3.3.2.5.6.3 Behavior.

The *open* operation shall open a file based upon the input fileName. The readOnly parameter indicates if the file should be opened for read access only. When readOnly is false the file is opened for write access.

## 3.1.3.3.2.5.6.4 Returns.

A *File* component shall be returned on successful completion of the *open* operation. A null *File* component reference shall be returned if the *open* operation is unsuccessful. If the file is opened with the readOnly flag set to true, then writes to the file will be considered an error.

## 3.1.3.3.2.5.6.5 Exceptions/Errors.

The InvalidFilename exception shall be raised when the filename is not valid.

The FileException shall be raised if the File does not exist or another file error occurred.

3.1.3.3.2.5.7 *mkdir*.

## 3.1.3.3.2.5.7.1 Brief Rationale.

The *mkdir* operation provides the ability to create a directory on the file system.

## 3.1.3.3.2.5.7.2 Synopsis.

```
void mkdir(in string directoryName) raises( InvalidFileName, FileException );
```

### 3.1.3.3.2.5.7.3 Behavior.

The *mkdir* operation shall create a *FileSystem* directory based on the *directoryName* given. The *mkdir* operation shall create all parent directories required to create the directory path given.

### 3.1.3.3.2.5.7.4 Returns.

None.

### 3.1.3.3.2.5.7.5 Exceptions/Errors.

The *InvalidFilename* exception shall be raised when the directory name is not valid.

The *FileException* shall be raised if a file-related error occurred during the operation.

### 3.1.3.3.2.5.8 *rmdir*.

#### 3.1.3.3.2.5.8.1 Brief Rationale.

The *rmdir* operation provides the ability to remove a directory from the file system.

#### 3.1.3.3.2.5.8.2 Synopsis.

```
void rmdir(in string directoryName) raises( InvalidFileName, FileException );
```

#### 3.1.3.3.2.5.8.3 Behavior.

The *rmdir* operation shall remove a *FileSystem* directory based on the *directoryName* given.

#### 3.1.3.3.2.5.8.4 Returns.

None.

#### 3.1.3.3.2.5.8.5 Exceptions/Errors.

The *InvalidFilename* exception shall be raised when the directory name is not valid.

The *FileException* shall be raised if the Directory does not exist or another file-related error occurred.

### 3.1.3.3.2.5.9 *query*.

#### 3.1.3.3.2.5.9.1 Brief Rationale.

The *query* operation provides the ability to retrieve information about a file system.

#### 3.1.3.3.2.5.9.2 Synopsis.

```
void query(inout Properties fileSystemProperties) raises(
UnknownFileSystemProperties );
```

#### 3.1.3.3.2.5.9.3 Behavior.

The *query* operation shall return file system information to the calling client based upon the given *fileSystemProperties*' ID.

As a minimum, the following *fileSystemProperties* shall be supported:

1. *SIZE* - an ID value of "SIZE" causes query to return an unsigned Long containing the file system size (in octets).
2. *AVAILABLE\_SPACE* - an ID value of "AVAILABLE\_SPACE" causes the *query* operation to return an unsigned Long containing the available space on the file system (in octets).

#### 3.1.3.3.2.5.9.4 Returns.

None.

#### 3.1.3.3.2.5.9.5 Exceptions/Errors.

The `UnknownFileSystemProperties` exception shall be raised when the given file system property is not recognized by the *query* operation.

### 3.1.3.3.3 *FileManager*.

#### 3.1.3.3.3.1 Description.

Multiple, distributed *FileSystems* may be accessed through a *FileManager*. The *FileManager* interface appears to be a single *FileSystem* although the actual file storage may span multiple physical file systems.

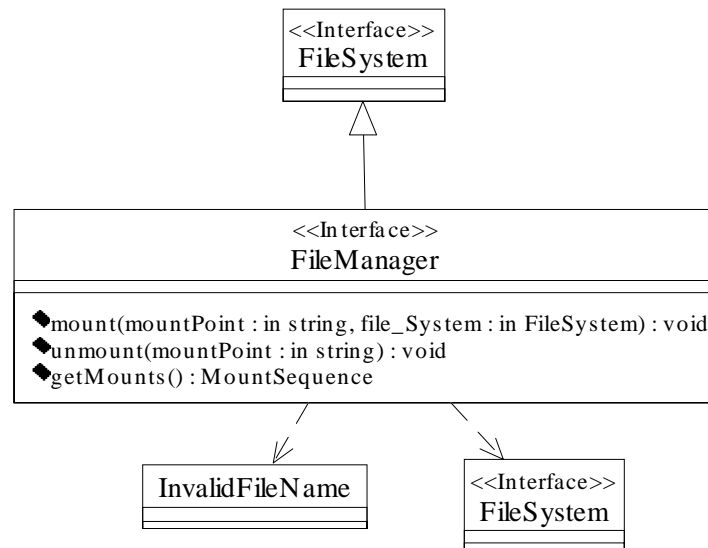
This is called a federated file system. A federated file system is created using the *mount* and *unmount* operations. Typically, the *DomainManager* or system initialization software will invoke these operations.

The *FileManager* inherits the IDL interface of a *FileSystem*. Based upon the pathname of a directory or file and the set of mounted filesystems, the *FileManager* will delegate the *FileSystem* operations to the appropriate *FileSystem*. For example, if a *FileSystem* is mounted at `/ppc2`, an *open* operation for a file called `/ppc2/profile.xml` would be delegated to the mounted *FileSystem*. The mounted *FileSystem* will be given the filename relative to it. In this example the *FileSystem*'s *open* operation would receive `/profile.xml` as the `fileName` argument.

Another example of this concept can be shown using the *copy* operation. When a client invokes the *copy* operation, the *FileManager* will delegate operations to the appropriate *FileSystems* (based upon supplied pathnames) thereby allowing copy of files between *FileSystems*.

If a client does not need to mount and unmount *FileSystems*, it can treat the *FileManager* as a *FileSystem* by CORBA widening a *FileManager* reference to a *FileSystem* reference. One can always widen a *FileManager* to a *FileSystem* since the *FileManager* is derived from a *FileSystem*.

## 3.1.3.3.3.2 UML.

**Figure 3-20. *FileManager* Interface UML**

## 3.1.3.3.3.3 Types.

## 3.1.3.3.3.3.1 MountType.

The MountType structure shall be used to identify the *FileSystems* mounted within the *FileManager*.

```

struct MountType {
    string mountPoint;
    FileSystem fs;
};
  
```

## 3.1.3.3.3.3.2 MountSequence.

The MountSequence is an unbounded sequence of Mount types.

```

typedef sequence<MountType> MountSequence;
  
```

## 3.1.3.3.3.3.3 FileSystemPropertyType.

The FileSystemPropertyType shall be used to associate a property with a specific *FileSystem*.

```

struct FileSystemPropertyType {
    string fileName;
    DataType property;
};
  
```

## 3.1.3.3.3.3.4 FileSystemPropertySequence.

The FileSystemPropertySequence is an unbounded sequence of FileSystemPropertyTypes.

```

typedef sequence<FileSystemPropertyType> FileSystemPropertySequence;
  
```

### 3.1.3.3.3.5 NonExistentMount.

```
exception NonExistentMount {};
```

This exception indicates a mount point does not exist within the *FileManager*.

### 3.1.3.3.3.6 MountPointAlreadyExists.

This exception indicates the mount point is already in use in the file manager.

```
exception MountPointAlreadyExists {};
```

### 3.1.3.3.3.7 InvalidFileSystem.

This exception indicates the *FileSystem* is a null (nil) object reference.

```
exception InvalidFileSystem {};
```

### 3.1.3.3.3.4 Attributes.

N/A.

### 3.1.3.3.3.5 Operations.

#### 3.1.3.3.3.5.1 *mount*.

##### 3.1.3.3.3.5.1.1 Brief Rationale.

The *FileManager* supports the notion of a federated file system. To create a federated file system, the *mount* operation associated a *FileSystem* with a mount point (a directory name).

##### 3.1.3.3.3.5.1.2 Synopsis.

```
void mount(in string mountPoint, in FileSystem fileSystem) raises(
InvalidFileName, InvalidFileSystem, MountPointAlreadyExists );
```

##### 3.1.3.3.3.5.1.3 Behavior.

The *mount* operation shall associate the specified *FileSystem* with the given *mountPoint*. The *mount* operation shall ensure that the *mountPoint* is a valid subdirectory path within the target *FileSystem*.

##### 3.1.3.3.3.5.1.4 Returns.

None.

##### 3.1.3.3.3.5.1.5 Exceptions/Errors.

The *NonExistentMount* exception shall be raised when the mount point (directory) name is not valid.

The *MountPointAlreadyExists* exception shall be raised when the mount point already exists in the file manager.

The *InvalidFileSystem* exception shall be raised when input *FileSystem* is a null object reference.

#### 3.1.3.3.3.5.2 *unmount*.

##### 3.1.3.3.3.5.2.1 Brief Rationale.

Mounted *FileSystems* may need to be removed from a *FileManager*.

##### 3.1.3.3.3.5.2.2 Synopsis.

```
void unmount(in string mountPoint) raises( NonExistentMount );
```

### 3.1.3.3.3.5.2.3 Behavior.

The *unmount* operation shall remove a mounted *FileSystem* from the *FileManager* whose mounted name matches the input mountPoint name.

### 3.1.3.3.3.5.2.4 Returns.

None.

### 3.1.3.3.3.5.2.5 Exceptions/Errors.

The NonexistentMount exception shall be raised when the mount point does not exist.

### 3.1.3.3.3.5.3 *getMounts*.

#### 3.1.3.3.3.5.3.1 Brief Rationale.

File management user interfaces may need to list a *FileManager*'s mounted *FileSystems*.

#### 3.1.3.3.3.5.3.2 Synopsis.

```
MountSequence getMounts();
```

#### 3.1.3.3.3.5.3.3 Behavior.

The *getMounts* operation shall return a sequence of Mount structures that describe the mounted *FileSystems*.

#### 3.1.3.3.3.5.3.4 Returns.

The *getMounts* operation returns a sequence of Mount structures.

#### 3.1.3.3.3.5.3.5 Exceptions/Errors.

None.

### 3.1.3.3.3.5.4 File System Operations.

The system may support multiple *FileSystem* implementations. Some *FileSystems* will correspond directly to a physical file system within the system. The *FileManager* interface shall support a federated, or distributed, file system that may span multiple *FileSystem* components. From the client perspective, the *FileManager* may be used just like any other *FileSystem* component since the *FileManager* inherits all the *FileSystem* operations.

The *FileManager*'s inherited *FileSystem* operations behavior shall implement the requirements of the *FileSystem* operations against the mounted file systems. The *FileSystem* operations shall ensure that the filename/directory arguments given are absolute pathnames relative to a mounted *FileSystem*.

### 3.1.3.3.3.5.5 *query*.

#### 3.1.3.3.3.5.5.1 Brief Rationale.

The inherited *query* operation provides the ability to retrieve the same information for a set of file systems.

#### 3.1.3.3.3.5.5.2 Synopsis.

```
void query(inout Properties fileSystemProperties) raises(
UnknownFileSystemProperties );
```

### 3.1.3.3.5.5.3 Behavior.

The *query* operation shall return file systems information to the calling client based upon the given *fileSystemProperties*' *fileName* and ID. If the *fileName* property is null, the given property for all of the mounted file systems shall be returned.

As a minimum, the following *fileSystemProperties* shall be supported:

1. SIZE - an ID value of "SIZE" causes *query* to return a value of a *FileSystemPropertySequence* when each item is a mounted file system name with a property value of unsigned Long containing the file system size (in octets).
2. AVAILABLE\_SPACE - an ID value of "AVAILABLE\_SPACE" causes *query* to return a value of a *FileSystemPropertySequence* when each item is a mounted file system name with a property value of unsigned Long containing the available space on the file system (in octets).

### 3.1.3.3.5.5.4 Returns.

None.

### 3.1.3.3.5.5.5 Exceptions/Errors.

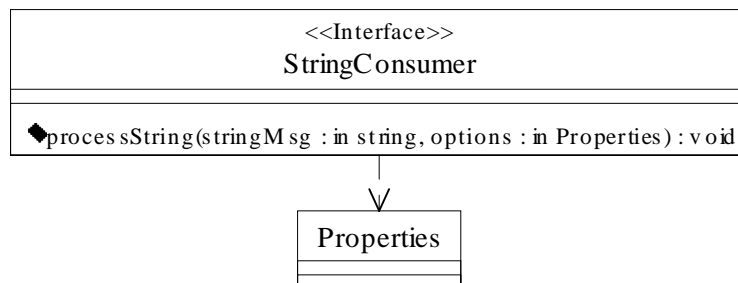
The *UnknownFileSystemProperties* exception shall be raised when the given file system property is not recognized by the *query* operation.

## 3.1.3.3.4 *StringConsumer*.

### 3.1.3.3.4.1 Description.

This interface is implemented by the *Logger* to push a string to consumers. The operations contained within this interface are used to define the consumers to whom producers are responsible for pushing messages.

### 3.1.3.3.4.2 UML.



**Figure 3-21. *StringConsumer* Interface UML**

### 3.1.3.3.4.3 Types.

N/A.

### 3.1.3.3.4.4 Attributes.

N/A.



### 3.1.3.3.4.5 Operations.

#### 3.1.3.3.4.5.1 *processString*.

##### 3.1.3.3.4.5.1.1 Brief Rationale.

*Logger* requires the *processString* operation in order to transfer CORBA string data to components implementing this interface (i.e. HCI).

##### 3.1.3.3.4.5.1.2 Synopsis.

```
oneway void processString (in string stringMsg, in Properties options);
```

##### 3.1.3.3.4.5.1.3 Behavior.

The implementation of this operation is component dependent.

##### 3.1.3.3.4.5.1.4 Returns.

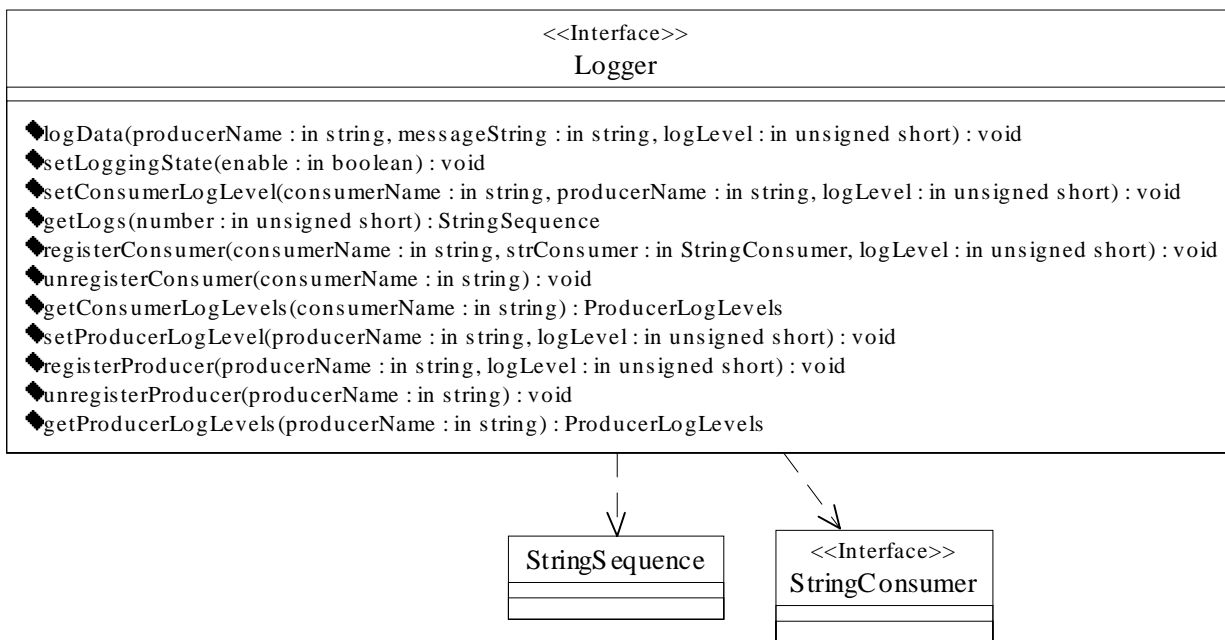
This operation does not return a value.

### 3.1.3.3.5 *Logger*.

#### 3.1.3.3.5.1 Description.

*Logger* defines the interface for logging data and receiving log data. The *Logger* interface is used to capture alarms, log warnings, and information messages and for pushing log messages to registered consumers. The interface provides operations for both producer and consumer clients. Consumers use the *StringConsumer* interface for receiving log data from a *Logger*. The *Logger* pushes log data to consumers based upon each consumer's log level.

#### 3.1.3.3.5.2 UML.



**Figure 3-22. *Logger* Interface UML**

### 3.1.3.3.5.3 Types.

The logLevel type contains the valid levels for log data being logged and logs data being received.

The logLevel shall be an unsigned short (16 bits) and is bitmapped 00 00 - 7F FF (hex). The MSB (d15) is a control bit to allow for log level manipulation. *Logger* Level manipulation using the control bit is as follows:

1. BIT Matching - When the control bit is one (1), then logging or forwarding log data shall be performed when the input log level matches registered consumer or producer log level. Example - LogLevel = C010 h (1100 0000 0001 0000 b) indicates only levels 14 and 4 are to be sent to a consumer or logged for a producer.
2. BIT Leveling - When the control bit is zero then logging or forwarding log data shall be performed when the input log level is less than or equal to registered consumer or producer log level. Example - LogLevel = 000A h indicates levels 9 (10 least significant bits) and below will be sent to a consumer or logged for a producer, and bits 4-14 are unused.

The logLevels can be set for consumers and producers. For producers the log level indicates the kind of information being logged. For consumers the log level determines what log information is sent to a consumer.

The *Logger*, consumers, and producers shall use the log levels in the following table. The log levels in table 3-1 are listed in order of significance. Log level value 0x0001 is of the highest significance. Each increasing log level thereafter decreases in significance. An event may cause more than one (1) log message.

**Table 3-1. *Logger, Consumers, and Producers Log Levels***

<b>LogLevel Numeric Value</b>	<b>LogLevel Name</b>	<b>Bit Mask Value</b>	<b>Description</b>
1	Security_Alarm	0x0001	This level indicates a security violation has occurred.
2	Failure_Alarm	0x0002	This level indicates faulted (disabled) operational behavior by either a software or hardware component.
3	Degraded_Alarm	0x0004	This level indicates degraded operational behavior by either a software or hardware component.
4	Reserved_1	0x0008	Reserved for future use.
5	Reserved_2	0x0010	Reserved for future use.
6	Exception_Error	0x0020	This level indicates an exception or abnormal condition has occurred.
7	Flow_Control_Error	0x0040	This level indicates a data flow control error has occurred.
8	Range_Error	0x0080	This level indicates a range or constraint error has occurred.
9	Usage_Error	0x0100	This level indicates a function was not performed due to improper configuration or state. This level can be used at startup or during runtime of a software or hardware component.
10	Reserved_3	0x0200	Reserved for future use.
11	Administrative_Event	0x0400	This level indicates an Administrative event (state change) has occurred (e.g. a software or hardware component has been commanded to offline (locked) state).
12	Statistic_Report	0x0800	This level indicates performance statistics that can be logged by a software component.
13	Reserved_4	0x1000	Reserved for future use.
14	Programmer_Debug1	0x2000	This level indicates programmer debug information containing normal processing flow information. The amount of information shall be kept to a minimum.
15	Programmer_Debug2	0x4000	This level indicates programmer debug information containing increased debug detail and amount (e.g. buffer / queue / memory dumps).
NA	Log_Control_Bit	0x8000	This level indicates how the log level data is to be used (BIT Matching or BIT Leveling as described above).

### 3.1.3.3.5.3.1 NameNotFound Exception.

This exception indicates the input name (producer or consumer) does not exist in the *Logger*.

```
exception NameNotFound {};
```

### 3.1.3.3.5.3.2 MATCH\_ALL\_NAMES.

This constant defines a special name that indicates all producer names are to be used by a consumer or for getting producer log levels.

```
constant string MATCH_ALL_NAMES = "*";
```

### 3.1.3.3.5.3.3 Producer Log Level Type.

```
struct ProducerLogLevelType
{
    string          name;
    unsigned short logLevel;
};
```

### 3.1.3.3.5.3.4 Producer Log Levels.

This type defines an unbounded sequence of producer log levels.

```
Typedef sequence <ProducerLogLevelType> ProducerLogLevels;
```

### 3.1.3.3.5.4 Attributes.

N/A.

### 3.1.3.3.5.5 Operations.

#### 3.1.3.3.5.5.1 *logData*.

##### 3.1.3.3.5.5.1.1 Brief Rationale.

Applications require the *logData* operation in order to log messages locally (in memory) and push log messages to registered consumers.

##### 3.1.3.3.5.5.1.2 Synopsis.

```
oneway void logData(in string producerName, in string messageString, in
unsigned short logLevel);
```

##### 3.1.3.3.5.5.1.3 Behavior.

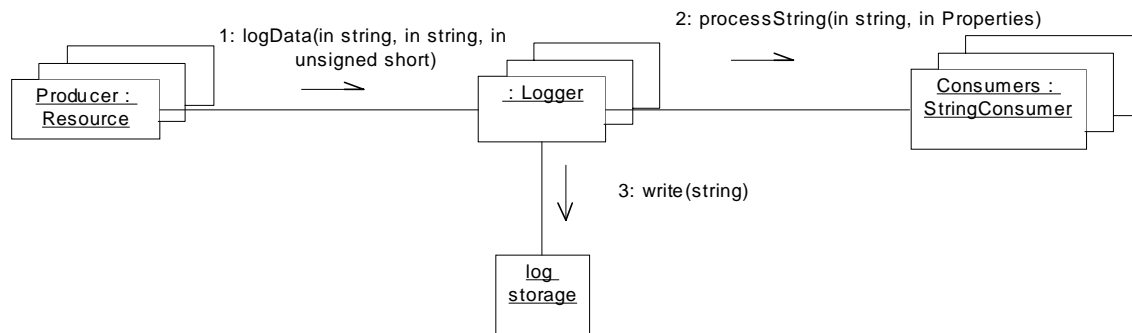
The *logData* operation (figure 3-23) shall only log messages locally and sends messages to consumers when the *Logger's* logging state is enabled.

The *logData* operation shall create a log message based upon input messageString, current time, input logLevel and input producerName.

The log message shall be saved locally in memory when the input logLevel criteria matches one of the registered producer's log levels.

The *LogData* operation shall push the String to registered consumers when the input log level criteria matches the registered consumers' log levels. A registered consumer's logLevel criteria of producerName set to "\*" shall cause all logging of data to be sent to this consumer if the input logLevel matches consumer log level criteria. A registered consumer's logLevel criteria of producerName beginning with characters and ended by "\*" shall cause all logging of data to be

sent to this consumer when the input producerName begins with the same characters and input logLevel matches consumer log level criteria. The *LogData* operation shall use the *StringConsumer::processString* operation for pushing messages to registered consumers. Current time, input logLevel and input producerName shall be placed in the *StringConsumer::processString* options parameter.



**Figure 3-23. LogData Operational Behavior**

#### 3.1.3.3.5.5.1.4 Returns.

The IDs and format values returned in the *StringConsumer::processString* options shall be:

1. ProducerName – The ID is 1 and the value is a CORBA string.
2. LogLevel – The ID is 2 and the value is CORBA unsigned short.
3. Time – The ID is 3 and the value is a CORBA string of form “HH:MM”.

#### 3.1.3.3.5.5.1.5 Exceptions/Errors.

N/A.

#### 3.1.3.3.5.5.2 setLoggingState.

##### 3.1.3.3.5.5.2.1 Brief Rationale.

This operation allows for controlling the overall logging state for the *Logger*.

##### 3.1.3.3.5.5.2.2 Synopsis.

```
void setLoggingState(in boolean enable);
```

##### 3.1.3.3.5.5.2.3 Behavior.

The *setLoggingState* operation shall set the *Logger*’s logging state to either allow or disallow the logging of all messages to console and file, and sending messages to consumers.

##### 3.1.3.3.5.5.2.4 Returns.

N/A.

##### 3.1.3.3.5.5.2.5 Exceptions/Errors.

N/A.

### 3.1.3.3.5.5.3 *setProducerLogLevel*.

#### 3.1.3.3.5.5.3.1 Brief Rationale.

Applications require the *logData* operation in order to set the log level for a producer, which controls the type of information being logged for this producer.

#### 3.1.3.3.5.5.3.2 Synopsis.

```
void setProducerLogLevel(in string producerName, in unsigned short logLevel)
raises ( NameNotFound );
```

#### 3.1.3.3.5.5.3.3 Behavior.

The *setProducerLogLevel* operation shall set the log level for a producer component. The *Logger* will use this log level to determine when to log messages for this producer.

#### 3.1.3.3.5.5.3.4 Returns.

N/A.

#### 3.1.3.3.5.5.3.5 Exceptions/Errors.

The *NameNotFound* exception shall be raised when the producer does not exist in the logger.

### 3.1.3.3.5.5.4 *setConsumerLogLevel*.

#### 3.1.3.3.5.5.4.1 Brief Rationale.

Applications require the *logData* operation in order to set the log level criteria for a consumer, which controls the type of information being pushed to this consumer.

#### 3.1.3.3.5.5.4.2 Synopsis.

```
void setConsumerLogLevel(in string consumerName, in string producerName, in
unsigned short logLevel) raises ( NameNotFound );
```

#### 3.1.3.3.5.5.4.3 Behavior.

The *setConsumerLogLevel* operation shall set the log level criteria for a consumer component based upon the input parameters. The consumer logLevel criteria is a set of logLevels per producer. This allows different producer log data to be pushed to consumers.

A special producerName of "\*" will cause all logging of data to be sent to this consumer if the input logLevel matches consumer log level.

A special producerName beginning with characters and ending in "\*" will cause all logging of data to be sent to this consumer when the input producer has the same beginning characters in their name and the input logLevel matches the consumer log level.

The *Logger* will use these log level criteria to determine when a producer-logged data is sent to this consumer.

#### 3.1.3.3.5.5.4.4 Returns.

N/A.

#### 3.1.3.3.5.5.4.5 Exceptions/Errors.

The *NameNotFound* exception shall be raised when the producer does not exist in the *Logger*.

### 3.1.3.3.5.5.5 *getLogs*.

#### 3.1.3.3.5.5.5.1 Brief Rationale.

Applications require the *getDisplaylast* operation in order to display a specified number of previous log messages stored locally.

#### 3.1.3.3.5.5.5.2 Synopsis.

```
Message::StringSequence getLogs(in unsigned short number);
```

The 'number' input parameter represents the last number of previous log entries stored locally.

#### 3.1.3.3.5.5.5.3 Behavior.

The *getLogs* operation shall return a qualified number of *Logger* messages stored locally, starting with the most recent. The number of log messages returned shall be based upon the input number as permitted by the actual size of the locally stored log messages.

#### 3.1.3.3.5.5.5.4 Returns.

This operation returns a sequence of character strings consisting of *Logger* messages.

#### 3.1.3.3.5.5.5.5 Exceptions/Errors.

N/A.

### 3.1.3.3.5.5.6 *registerConsumer*.

#### 3.1.3.3.5.5.6.1 Brief Rationale.

Applications require the *registerConsumer* operation in order to receive log messages that are being logged.

#### 3.1.3.3.5.5.6.2 Synopsis.

```
void registerConsumer(in string consumerName, in StringConsumer MsgConsumer,  
in unsigned short logLevel);
```

The *consumerName* identifies the consumer. The *MsgConsumer* is the *StringConsumer* component to be used for pushing log messages to the consumer. The *logLevel* initially identifies the kind of log information the consumer is interested in from all producers.

#### 3.1.3.3.5.5.6.3 Behavior.

The *registerConsumer* operation shall register a consumer with *Logger* using the input parameters. If the consumer already exists, the *registerConsumer* operation shall ignore the request. The *logLevel* shall initially be used for all log data being logged.

#### 3.1.3.3.5.5.6.4 Returns.

N/A.

#### 3.1.3.3.5.5.6.5 Exceptions/Errors.

N/A.

### 3.1.3.3.5.5.7 *registerProducer*.

#### 3.1.3.3.5.5.7.1 Brief Rationale.

Applications require the *registerProducer* operation in order to have their log information saved locally or to a file, provided the *logLevel* matches.

### 3.1.3.3.5.5.7.2 Synopsis.

```
void registerProducer(in string producerName, in unsigned short logLevel);
```

The *producerName* identifies the producer. The *logLevel* initially identifies the kind of log information required from the producer.

### 3.1.3.3.5.5.7.3 Behavior.

The *registerProducer* operation shall register a producer with *Logger* using the input parameters. If the producer already exists, the *registerProducer* operation shall ignore the request.

### 3.1.3.3.5.5.7.4 Returns.

N/A.

### 3.1.3.3.5.5.7.5 Exceptions/Errors.

N/A.

## 3.1.3.3.5.5.8 *unregisterConsumer*.

### 3.1.3.3.5.5.8.1 Brief Rationale.

Applications require the *unregisterConsumer* operation in order to remove a consumer from a logger, so messages are no longer pushed to the consumer.

### 3.1.3.3.5.5.8.2 Synopsis.

```
void unregisterConsumer(in string consumerName) raises ( NameNotFound );
```

### 3.1.3.3.5.5.8.3 Behavior.

The *unregisterConsumer* operation shall unregister a consumer from the *Logger* causing the *Logger* to stop sending log messages to a previously registered consumer.

### 3.1.3.3.5.5.8.4 Returns.

N/A.

### 3.1.3.3.5.5.8.5 Exceptions/Errors.

The *NameNotFound* exception shall be raised when the consumer does not exist in the *Logger*.

## 3.1.3.3.5.5.9 *unregisterProducer*.

### 3.1.3.3.5.5.9.1 Brief Rationale.

Applications require the *unregisterProducer* operation in order to remove a producer from a *Logger*, so messages are no longer logged for a producer.

### 3.1.3.3.5.5.9.2 Synopsis.

```
void unregisterProducer(in string producerName) raises ( NameNotFound );
```

### 3.1.3.3.5.5.9.3 Behavior.

The *unregisterProducer* operation shall unregister a producer from the *Logger* causing the *Logger* to stop all logging activities for that particular producer.

### 3.1.3.3.5.5.9.4 Returns.

N/A.



### 3.1.3.3.5.5.9.5 Exceptions/Errors.

The `NameNotFound` exception shall be raised when the producer does not exist in the *Logger*.

### 3.1.3.3.5.5.10 *getProducerLogLevels*.

#### 3.1.3.3.5.5.10.1 Brief Rationale.

This operation is required as a feedback mechanism to registered consumers for registered consumer(s) log levels.

#### 3.1.3.3.5.5.10.2 Synopsis.

```
ProducerLogLevels getProducerLogLevels(in string producerName);
```

#### 3.1.3.3.5.5.10.3 Behavior.

The *getProducerLogLevels* operation returns the current log levels for passed in registered producer component name(s). A special `ProducerName` of "\*" shall return all producers logLevels. A special `ProducerName` beginning with characters followed "\*" shall return all registered producers that have the same beginning characters in their name.

Examples:

"HQ" would be all registered producer names that match exactly with string "HQ".

"HQ\*" would be all registered producer names starting with string "HQ".

"\*" would be all registered producers.

#### 3.1.3.3.5.5.10.4 Returns.

This operation returns a sequence of producer name and associated log level data. The set returned may be empty when a `producerName` name match is not found.

#### 3.1.3.3.5.5.10.5 Exceptions/Errors.

N/A.

### 3.1.3.3.5.5.11 *getConsumerLogLevels*.

#### 3.1.3.3.5.5.11.1 Brief Rationale.

This operation is required as a feedback mechanism to registered consumers for a consumer's current log level.

#### 3.1.3.3.5.5.11.2 Synopsis.

```
ProducerLogLevels getConsumerLogLevels(in string consumerName);
```

#### 3.1.3.3.5.5.11.3 Behavior.

The *getConsumerLogLevels* operation returns the current log levels for passed in registered consumer component name(s).

#### 3.1.3.3.5.5.11.4 Returns.

This operation returns a sequence of producer name and associated log level data for a specified consumer. The set returned may be empty when a `consumerName` name match is not found.

#### 3.1.3.3.5.5.11.5 Exceptions/Errors.

N/A.

#### 3.1.3.3.6 *Timer*.

No SCA-mandated *Timer* interfaces have been defined at this time.

#### 3.1.3.4 Domain Profile.

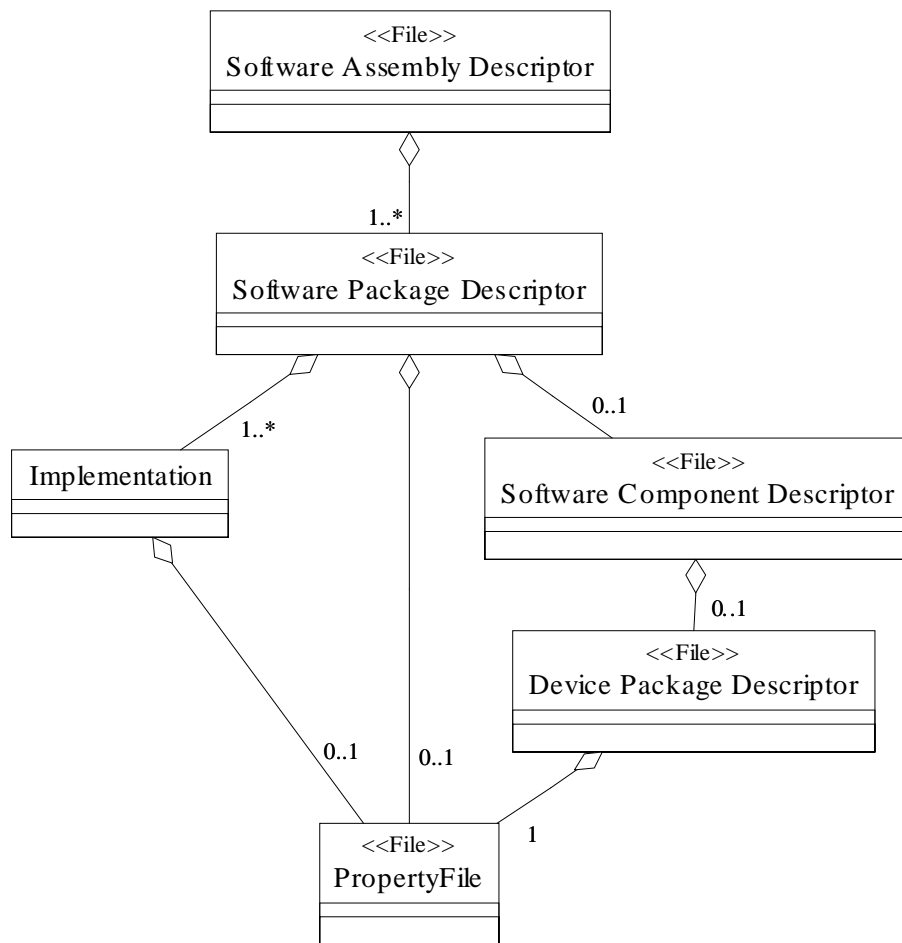
The hardware devices and software components that make up an SCA system domain are described by a set of files that are collectively referred to as a Domain Profile. These files describe the identity, capabilities, properties, inter-dependencies, and location of the hardware devices and software components that make up the system. All of the descriptive data about a system is expressed in the XML vocabulary. For purposes of this SCA specification, the elements of the XML vocabulary have been based upon the OMG's CORBA Components specification (orbos/99-07-01). [\[Note: At the time of this writing, 99-07-01 is a draft standard\].](#)

The types of XML files that are used to describe a system's hardware and software assets are depicted in figure 3-24. The XML vocabulary within each of these files describes a distinct aspect of the hardware and software assets.

Domain Profile files shall use the format of the Document Type Definitions (DTDs) provided in Appendix D.

---

<sup>3</sup> [“Design Patterns : Elements of Reusable Object-Oriented Software”](#) (Addison-Wesley Professional Computing) Gamma, Helm, Johnson, and Vlissides, pg. 139



**Figure 3-24. Relationship of Domain Profile XML File Types**

#### 3.1.3.4.1 Software Package Descriptor.

A Software Package Descriptor identifies a software package. A Software Package Descriptor file has a “.spd” extension. General information about a software package, such as the name, author, location of implementation (component and/or assembly files) and property files, and package level and/or implementation level hardware and/or software dependencies are contained in a Software Package Descriptor file. The Implementation element is part of the Software Package Descriptor.

#### 3.1.3.4.2 Software Component Descriptor.

A Software Component Descriptor contains information about a specific SCA software component (*Resource*, *ResourceFactory*, *Device*). A Software Component Descriptor file has a “.scd” extension. Information about the interfaces that a component provides and/or uses as well as any specific property files referenced by the component are contained in a Software Component Descriptor file.

#### 3.1.3.4.3 Software Assembly Descriptor.

A Software Assembly Descriptor contains information about the interface connections that are supported among two or more software components that make up a software assembly. A Software Assembly Descriptor file has a “.sad” extension.

#### 3.1.3.4.4 Property File.

A Property File contains information about the properties applicable to a software package or a software component. A Software Property File has a “.spf” extension. A Device Property File contains information about the properties of a device. A Device Property File has a “.dpf” extension. Information such as processor types, device capacities, and programmable devices are contained in a Device Property File.

#### 3.1.3.4.5 Device Package Descriptor File.

A Device Package Descriptor File identifies a class of a device. A Device Descriptor File has a “.dpd” extension. Device identification information, including the device name, device class, model number, and serial number are contained in a Device Descriptor File.

#### 3.1.3.4.6 Device Assembly Descriptor.

A Device Assembly Descriptor contains information about the interface connections that are supported among two or more hardware devices that make up a device assembly. A Device Assembly Descriptor file has a “.dad” extension. [The Device Assembly Descriptor has not been defined at this time.]

## 3.2 APPLICATIONS.

Applications are programs that perform the functions of a specific SCA-compliant product. They must meet the requirements of a procurement specification and are not defined by the SCA except as they interface to the OE.

### 3.2.1 General Application Requirements.

#### 3.2.1.1 OS Services.

Applications shall be limited to using the OS services that are designated as mandatory in the SCA AEP as specified in section 3.1.1.

Applications shall perform file access through the CF *File* interfaces.

To ensure controlled termination, applications shall have a signal handler installed for SIGQUIT.

#### 3.2.1.2 CORBA Services.

Applications shall be limited to using CORBA and CORBA services as specified in section 3.1.2. Use of Naming Services per 3.1.2.2.1 is optional; if it is not used, applications shall include stringified IORs in their Software Profile

#### 3.2.1.3 CF Interfaces.

Applications shall use the CF interfaces as specified in section 3.1.3.1 with the corresponding IDL in Appendix C, except as follows:

1. The use of *StringConsumer* per section 3.1.3.3.4 is optional if components do not consume *Logger* data.
2. The use of *Logger* per section 3.1.3.3.5 is optional if data is not logged.
3. The use of *ResourceFactory* per section 3.1.3.1.6 is optional.

Each application process that uses Naming Service shall support the name parameters passed by the *DeviceManager* (/ DomainName / NodeName / [other context sequences] / ComponentName\_UniqueIdentifier). This registration shall be placed underneath the last naming context passed to the application. (In the naming parameter string, each "slash" (/) represents a separate naming context.)

Applications' components and *DeviceManagers* shall be provided with Domain Profile files per 3.1.3.4. These files shall be in XML, using the format shown in Appendix D.

#### 3.2.1.3.1 CF Interface Extensions.

Applications are allowed to extend the CF *Port* interfaces (section 3.1.3.1.1) and the CF *Resource* interfaces (section 3.1.3.1.5). Extensions to the CF *Port* interface define the *Port* data interfaces. Extensions to the CF *Resource* interface allow for the inheritance of additional component-specific interfaces. All extensions shall be documented in IDL and identified with a Universally Unique Identifier (UUID), as defined in section 7.4. This UUID shall be included in the Software Profile to identify the interface requirements of the application to the *DomainManager*.

The CF *Port* extensions for basic data sequence types and the push and pull interfaces for using these types are shown in Figure 3-25 and Figure 3-26. If the extensions are used, they shall be implemented using the IDL in the Appendix C.

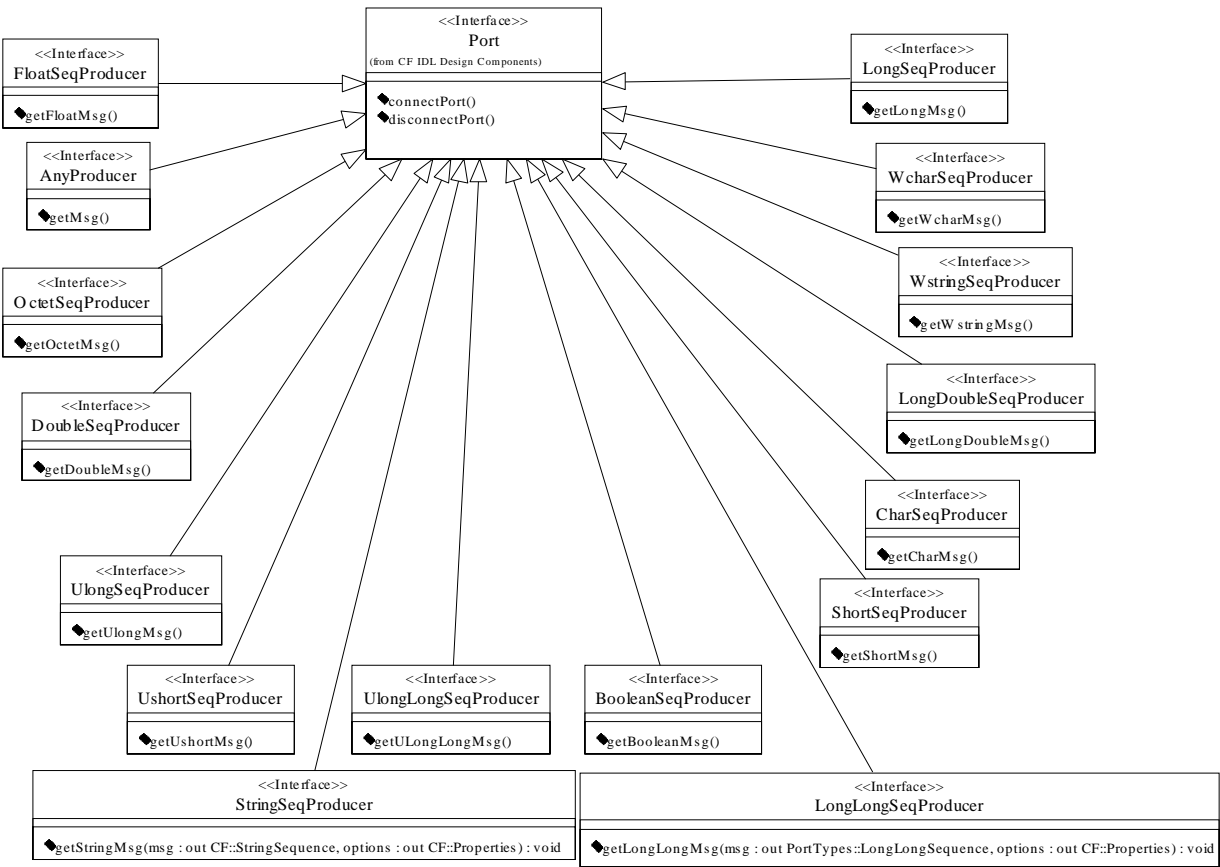


Figure 3-25. PushPort Data Interfaces

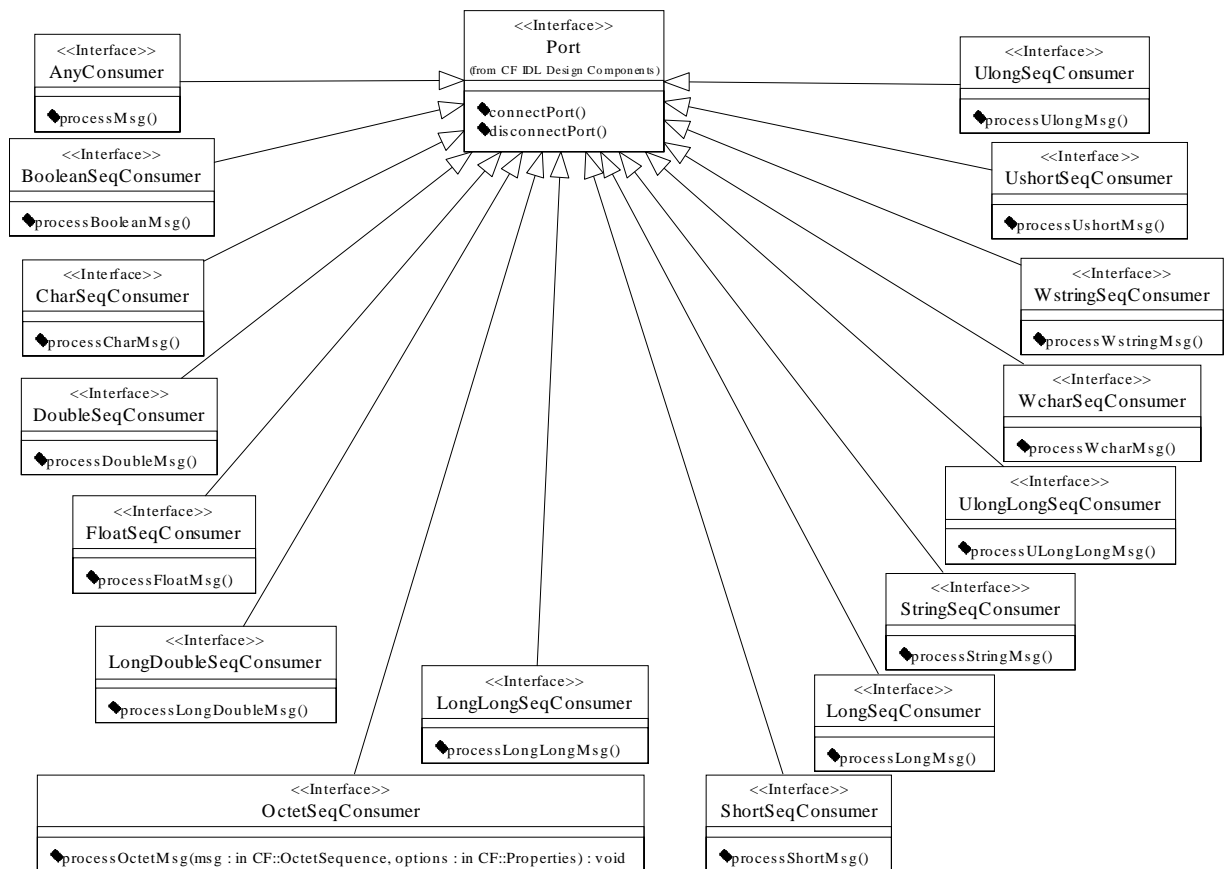


Figure 3-26. PullPort Data Interfaces

### 3.2.2 Application Interfaces.

Applications consist of multiple components. Component interfaces, other than CF interfaces, may be hidden from view by other applications or the CF; they are "wrapped" by components with visible interfaces. Applications' components' interfaces shall be visible and defined as described herein if:

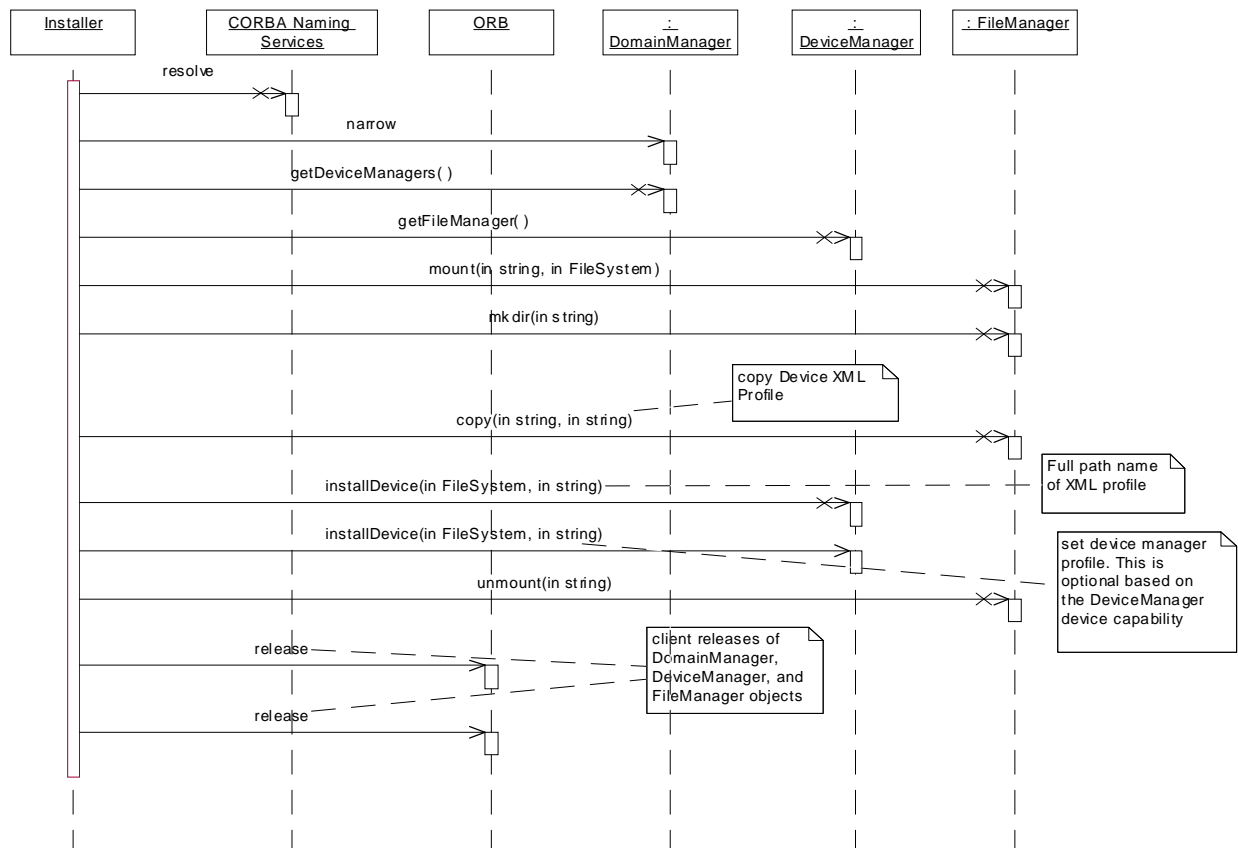
1. the component provides a service that is used by more than one application, or
2. the service user requires the interface to be common across service implementations.

#### 3.2.2.1 Utility Applications.

##### 3.2.2.1.1 Installer Utility.

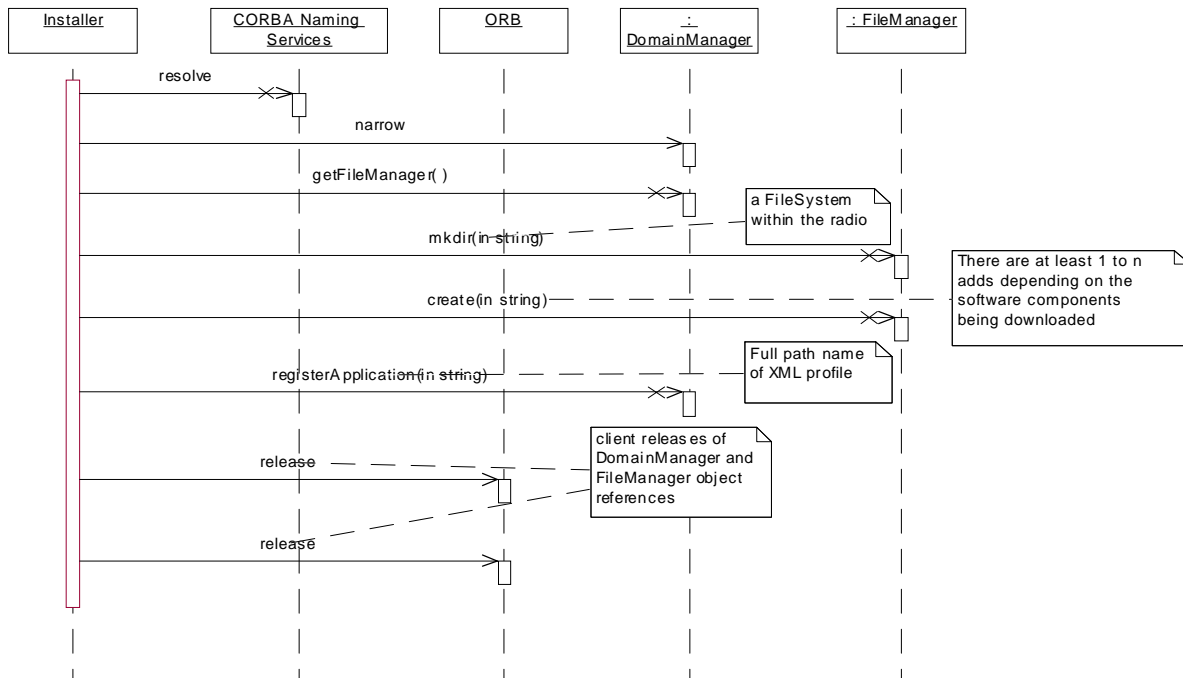
Installer is a generic name given to utility applications used for installing and uninstalling devices and components within the system. Installers shall use the CF *DomainManager*, *DeviceManager*, and *FileManager* interfaces for those operations. Installers can reside inside or outside the system.

Figure 3-27 and figure 3-28 illustrate device and software installations using the CF.



**Figure 3-27. Device Installation Sequence Diagram**





**Figure 3-28. Software Installation Sequence Diagram**

### 3.2.2.2 Service APIs.

Service APIs provide definition and standardization of common functionality and interfaces for use by SCA applications (e.g. waveforms). Services include Network Services, Security Services, and I/O Services. Each Service API is defined by a Service Definition and Transfer Mechanism. The API Supplement to the SCA Specification provides additional details and requirements for Service APIs.

*{The API Supplement to the SCA Specification is currently being developed.}*

#### 3.2.2.2.1 Service Definitions.

SCA-compliant Service Definitions consist of APIs, behavior, state, priority and additional information that provide the contract between the Service Provider and the Service User. IDL is used to define the interfaces for Service Definitions to foster reuse and interoperability. IDL provides a method to inherit from multiple interfaces to form a new Service Definition.

##### 3.2.2.2.1.1 Format.

All SCA-compliant Service Definitions shall have their interfaces described in IDL, except as allowed in 3.2.2.2.1.5.

SCA-compliant Service Definitions shall conform to the Service Definition Description (SDD) provided in Appendix E, except as allowed in 3.2.2.2.1.5.

#### 3.2.2.2.1.2 Service Definition Usage and Creation.

The structure and language requirements of the Service Definitions have been selected to provide commonality between implementations to foster reuse and portability of applications. To further these ends, the following methods for use and creation of Service Definitions are presented.

1. Reuse an existing Service Definition: If an existing Service Definition is functionally identical to the new service's interface, or the new service can easily be mapped to an existing Service Definition, the existing Service Definition should be reused.
2. Create a new Service Definition by inheriting an existing Service Definition and then extending its features and capabilities.
3. Translate the existing interface of the service to IDL to create a new Service Definition.
4. Create a new Service Definition: If the service being implemented has an interface that can not be defined by existing Service Definitions, then a new Service Definition needs to be created using the SDD.

(For these identified methods, it is recommended that the order of preference flow from Item #1 to Item #4.)

New (and extensions to existing) Service Definitions should use the Building Blocks described in 3.2.2.2.1.6 for interfaces that are common to the new definition.

#### 3.2.2.2.1.3 Service Definition Identification.

Each Service Definition shall be identified by a UUID as defined in section 7.4.

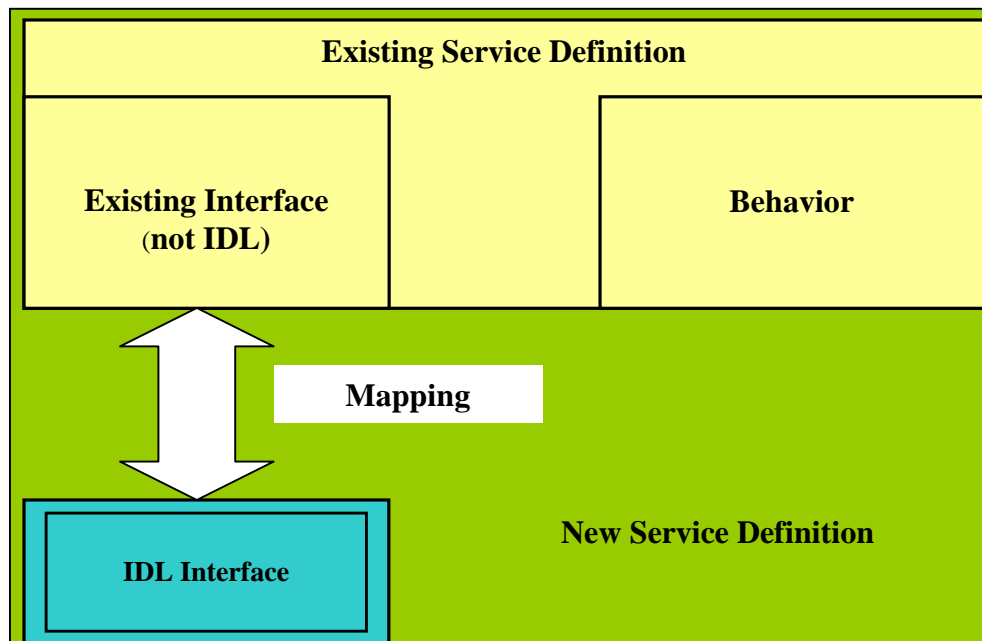
#### 3.2.2.2.1.4 Registration.

Public (i.e. non-proprietary) service interfaces used in SCA-compliant systems shall have their public Service Definitions and associated UUIDs registered as defined in section 7.4.

Private (i.e. proprietary) service interfaces used in SCA-compliant systems shall have their private Service Definition UUIDs, with a public description of the API, registered. They are allowed, but not required, to register their private Service Definitions with the Registration Body. [Note: it is a requirement for JTRS implementations of the SCA that Service Definitions be public, as described in the API Supplement to the SCA Specification.]

#### 3.2.2.2.1.5 Existing Service Definitions.

It is not the intent of this document to force creation of new documentation for existing Service Definitions that have commercial and/or government acceptance. Method 3 in section 3.2.2.2.1.2 allows the reuse of existing Service Definitions that do not have IDL interfaces by mapping an IDL interface to that Service Definition as shown in figure 3-29.



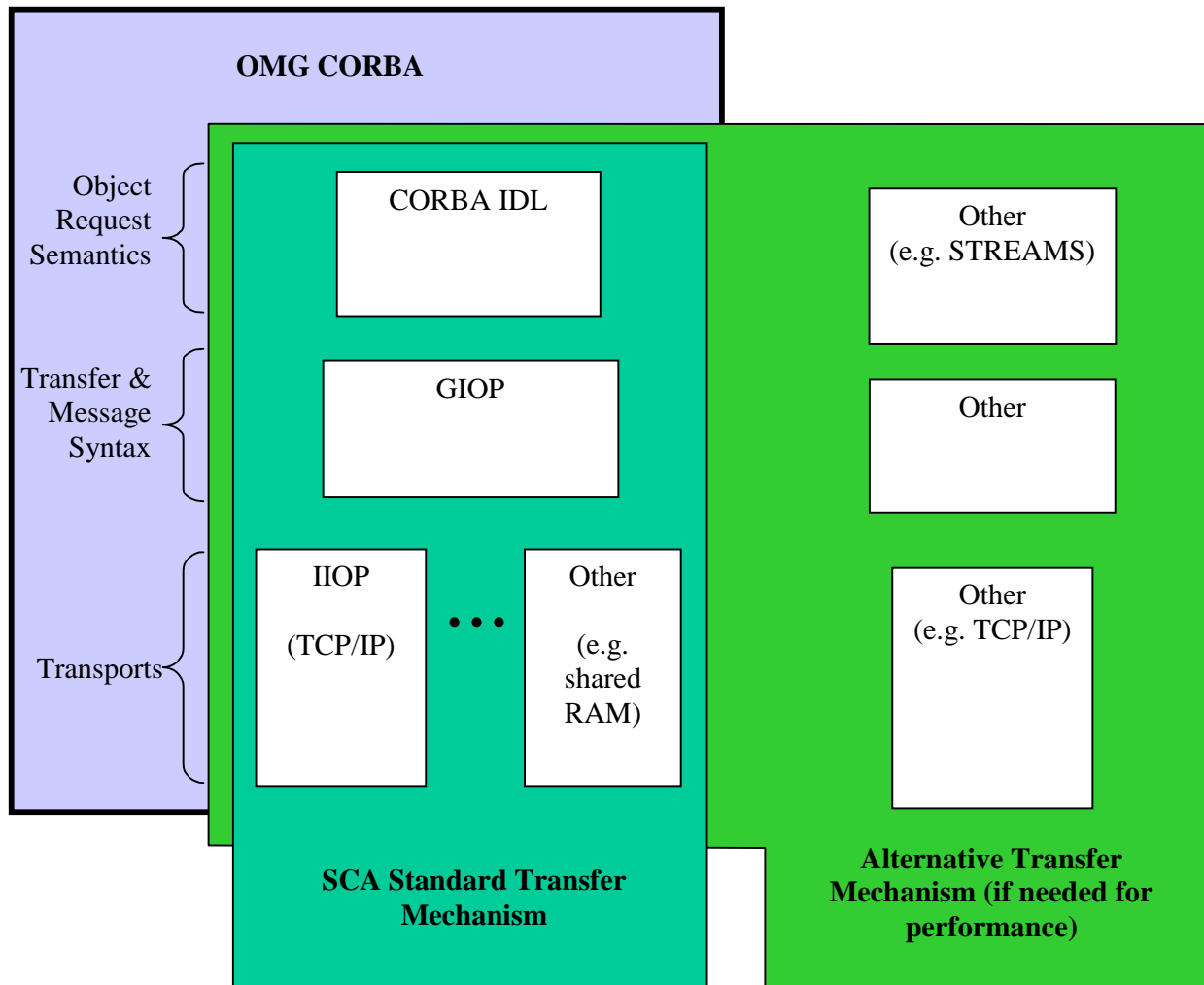
**Figure 3-29. Reusing an Existing Service Definition Without an IDL Interface**

#### 3.2.2.2.1.6 Service Definition Building Blocks.

As a further means to attain reuse and portability, common elements (interfaces with defined structure, behavior, and IDL) are provided in the API Supplement to the SCA Specification.

#### 3.2.2.2.2 API Transfer Mechanisms.

A Transfer Mechanism provides the communication between a service provider and a service user that may be co-located or distributed across different processors. Figure 3-30 shows the standard and alternate transfer mechanism structure for APIs.



**Figure 3-30. Standard and Alternate Transfer Mechanism**

#### 3.2.2.2.2.1 Standard Networking Transfer Mechanism.

The standard networking transfer mechanism shall be CORBA except as allowed in 3.2.2.2.2.2.

#### 3.2.2.2.2.2 Alternate Networking Transfer Mechanism.

An alternate networking transfer mechanism is allowed for the following case.

1. Application performance cannot be achieved with the standard transfer mechanism.

When an alternate transfer mechanism is used for real-time control and data flow, the transfer mechanism for initialization and non-real-time control shall use the standard transfer mechanism (if those controls can be separated).

When an alternate transfer mechanism is used, the transfer and message syntax of the alternate transfer mechanism shall be mapped to the IDL of the API Service Definition.

This mapping shall be identified by a UUID (separate from the Service Definition UUID).

The description of the alternate transfer mechanism, an analysis supporting the performance need for the alternate mechanism, the mappings to the Service Definition, and the associated UUIDs shall be registered as defined in section 7.4.

#### 3.2.2.2.2.1 API Instance Behavior.

Irrespective of the transfer mechanism used, all behavior including state transitions and priorities defined in the service definition shall be obeyed by a API Instance.

#### 3.2.2.2.2.2 Alternate Transfer Mechanism Standards.

Transfer mechanisms shall be in accordance with commercial or government standards.

#### 3.2.2.2.2.3 Alternate Transfer Mechanism Selection.

In addition to the above, transfer mechanism selection should consider the availability of supporting products that have wide usage, are available from multiple vendors, and are expected to have long-term support in the industry.

### **3.3 GENERAL SOFTWARE RULES.**

This section identifies those rules and recommendations specific to the Software Architecture that are not specifically addressed elsewhere in this specification.

#### **3.3.1 Software Development Languages.**

##### **3.3.1.1 New Software.**

Software developed for an SCA-compliant product shall be developed in a standard higher order language, except as provided below, for ease in processor portability. The goal of new development should be to provide SW that is independent from platform and environment details, ensuring minimal portability issues.

An exception is allowed to this requirement, if there are program performance requirements that require the use of assembly language programming.

##### **3.3.1.2 Legacy Software.**

Legacy software is not required to be rewritten in a standard higher order language. However legacy SW shall be interfaced to the core framework in accordance with this specification, through the use of Adapters if necessary.

## 4 HARDWARE ARCHITECTURE DEFINITION

This section describes the methodology of using the SCA as the basis for partitioning the Hardware (HW) Architecture in terms of an Object-Oriented approach. This Object-Oriented approach describes a hierarchy of hardware class and subclass objects that represent the architecture. Characteristics, or attributes, associated with each hierarchical class form the domain independent basis for the definition of each physical hardware device. Section 4.5 specifies the hardware requirements.

### 4.1 BASIC APPROACH.

The definition of the HW Architecture consists of a set of HW classes that are common across all domains. The top-level hardware classes correspond with top-level hardware functions. These top-level HW classes are further refined into subclasses that correspond with lower-level hardware functions. The attributes associated with these classes and/or subclasses describe the individual class or subclass contributions to system features and capabilities.

During implementation, this hardware class structure can be used to describe the hardware implementation in accordance with procurement specifications. This object-oriented approach enables a consistent application of the HW architecture (classes and rules) across the various domains (i.e., Handheld, Dismounted, Vehicular, Airborne, and Maritime/Fixed).

Attributes and the HW class structure will potentially have multiple users over the lifetime of each hardware module. Initially, when the radio system engineer is designing a radio system, class attributes provide a place to sort top level requirements, either by direct allocation or by analysis and allocation. After physical partitioning is performed, the attributes outline HW module(s) specification(s). The hardware designer, through the module specifications, in effect, uses the attributes to characterize the design of the modules.

Software applications also become users of HW attributes. The attributes are reported to the *DomainManager* through the Device Profiles. As software applications become more sophisticated, they will become increasingly dependent upon HW attributes, used potentially both as variables or in software dependency checks in the applications.

### 4.2 CLASS STRUCTURE.

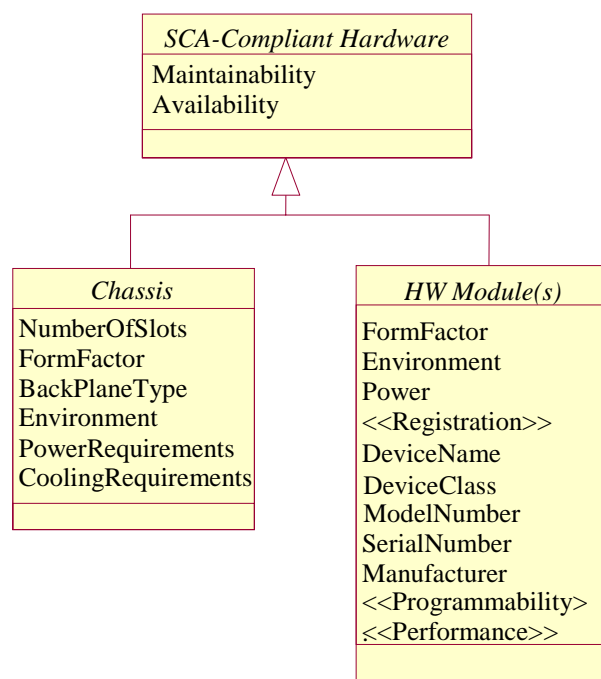
Class structure is the hierarchy that depicts how object-oriented classes and subclasses are related. The SCA hardware class structure identifies functional elements that in turn are used in the creation of physical system elements (HW devices). Using this object-oriented approach, devices "inherit" from the class structure and share common physical and interface attributes, thus making it easier to identify and compare device interchangeability. (In this use, the term "inherit" simply means that attributes at a higher class-level are common with all the subclasses. In the following figures, this feature is shown by a hollow arrow, the UML symbol for "generalization".)

Hardware devices represent physical implementations whose attributes are assigned specific values. In this sense, the attributes define domain-neutral class objects (abstract classes) and the values of the class attributes then place specific requirements on the implementation. HW devices inherit common attributes via the hardware class structure. Devices can then be developed to satisfy procurement-specific requirements. All hardware devices will have values

assigned to the class attributes. (The attributes shown in the figures in this section are representative of the attributes associated with the respective classes and are provided for illustrative purposes.)

#### 4.2.1 Top Level Class Structure.

At the system level, hardware conforms to the class structure depicted in figure 4-1 and figure 4-2. The top-level *SCA-Compliant Hardware* class defines the system procurement-associated attributes such as maintainability and availability requirements. The *Chassis* class has unique physical, interface, platform power and external environment attributes that are related to external factors rather than individual modules within the chassis. The *HWModule(s)* class represents a wide variety of SCA-compliant physical hardware. Subclasses of *HWModule(s)* inherit all its attributes, including those shown in figure 4-2. Stereotypes, indicated by enclosure in double brackets (<<stereotype>>), are included in the class diagrams to better group and manage attribute labels and titles. The stereotypes are generally associated with particular users of the attributes.



**Figure 4-1. Top Level Hardware Class Structure**

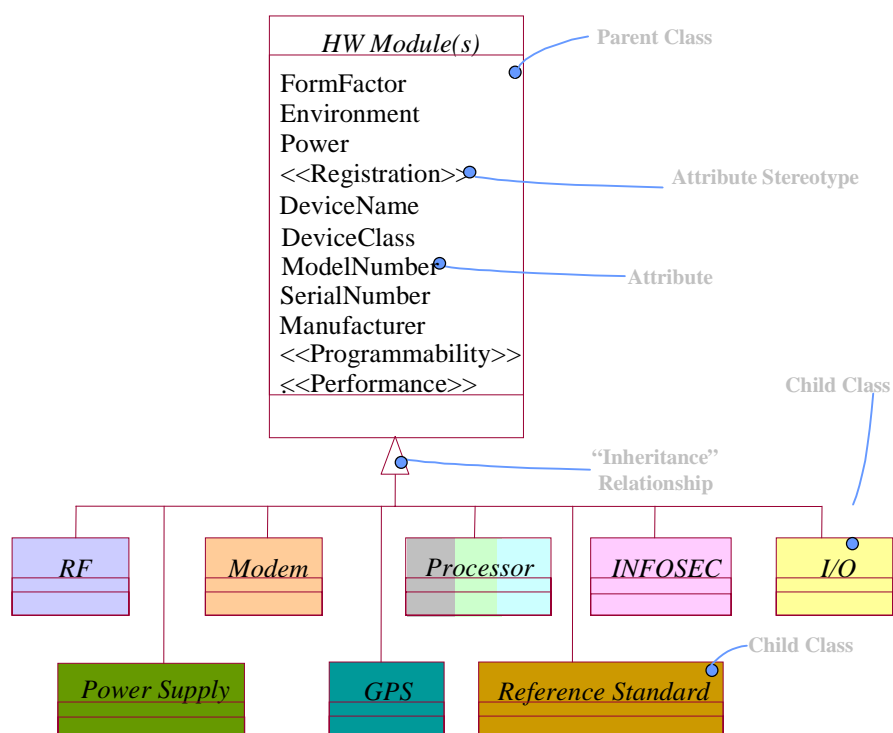
The *Chassis* subclass includes the attributes of number of module slots, form factor, back plane type, platform environmental, power and cooling requirements. The *HWModule(s)* class is the parent to all module sub-classes and provides the basic attributes that are inherited by all hardware modules. As the class structure hierarchy extends from the more general top level down into the more specific lower levels, each subclass inherits the attributes of all the preceding hierarchy of classes. Module compatibility can be ascertained by comparing appropriate instantiated attributes.



#### 4.2.2 *HWModule(s)* Class Structure.

The JTRS concepts of hardware reuse, extendibility and expandability dictate a modular implementation approach. The hardware architecture presents two very distinct module types. The first type contains software intensive processing elements (i.e., Digital Signal Processor (DSP) modules and General Purpose Processor (GPP) cards), while the second type contains non-programmable functionality (such as RF elements). As programmable capability and programmable hardware technologies evolve, functionality will migrate from totally embedded hardware towards more software intensive applications of the hardware functions.

There is a blurring of hardware/software functionality as systems are implemented. Functions are realized from a combination of both hardware embedded functions and software functions. Thus the *HWModule(s)* class framework shown in figure 4-2 includes functional classes that are strictly programmable in nature (*Processor*) and others that have embedded functionality. This provides the framework necessary to construct the elements for a software programmable radio.



**Figure 4-2. Hardware Module Class Structure**

The hardware class structure is expandable through the addition of new classes or through the addition of new attributes to existing classes to allow for future growth capabilities. Stereotypes, indicated by enclosure in double brackets (<<stereotype>>), are included in the class diagrams to better group and manage attribute labels and titles.

In the HW Module class, <<Registration>> attributes are those that become part of a Device Profile as reported through a Device Package Descriptor file. All other stereotypes indicate

attributes that, when reported, become part of the Device Profile as reported through a Property File.

#### 4.2.3 Class Structure with Extensions.

Each hardware class can be extended further to provide additional attribute granularity. This methodology provides both a formalized structure for hardware definition and the inherent flexibility needed to allow for evolving requirements as well as hardware and software capabilities.

##### 4.2.3.1 *RF* Class Extension.

The subclasses in figure 4-3 extend the *RF* class hierarchy. These subclasses relate to the typical range of RF hardware devices such as, Antennas, Receivers, Exciters, and Power Amplifiers. As with all HW subclasses, the attributes contained within these RF subclasses attempt to encapsulate the functionality that can be used to describe the unique mix of features and capabilities of the associated hardware device.

Cosite performance considerations place a special burden on the *RF* class. The intelligent management of cosite performance requires monitoring and control of many of the RF subclass modules. The hardware architecture supports cosite operation in two ways. First, there is a cosite sub-class. This encapsulates the hardware that is specifically provided for cosite operation. Second, a <<CositePerformance>> stereotype groups those attributes useful for a cosite manager application. Such an application, while not part of the architecture itself, is an implementation-specific capability to coordinates RF assets.

Antennas have historically been passive elements typically attached to the structure that houses the communications system. While remaining very domain and platform unique, technology growth continually improves the capabilities that can be performed in the communications system 'front end', necessitating the inclusion of antennas in the core of JTRS. "Smart" antennas include receive, transmit, and cosite mitigating elements, blurring the functional separation lines. For this reason and because of the key role that antenna systems play in cosite management, "Antenna" is incorporated in the class structure as an RF subclass.

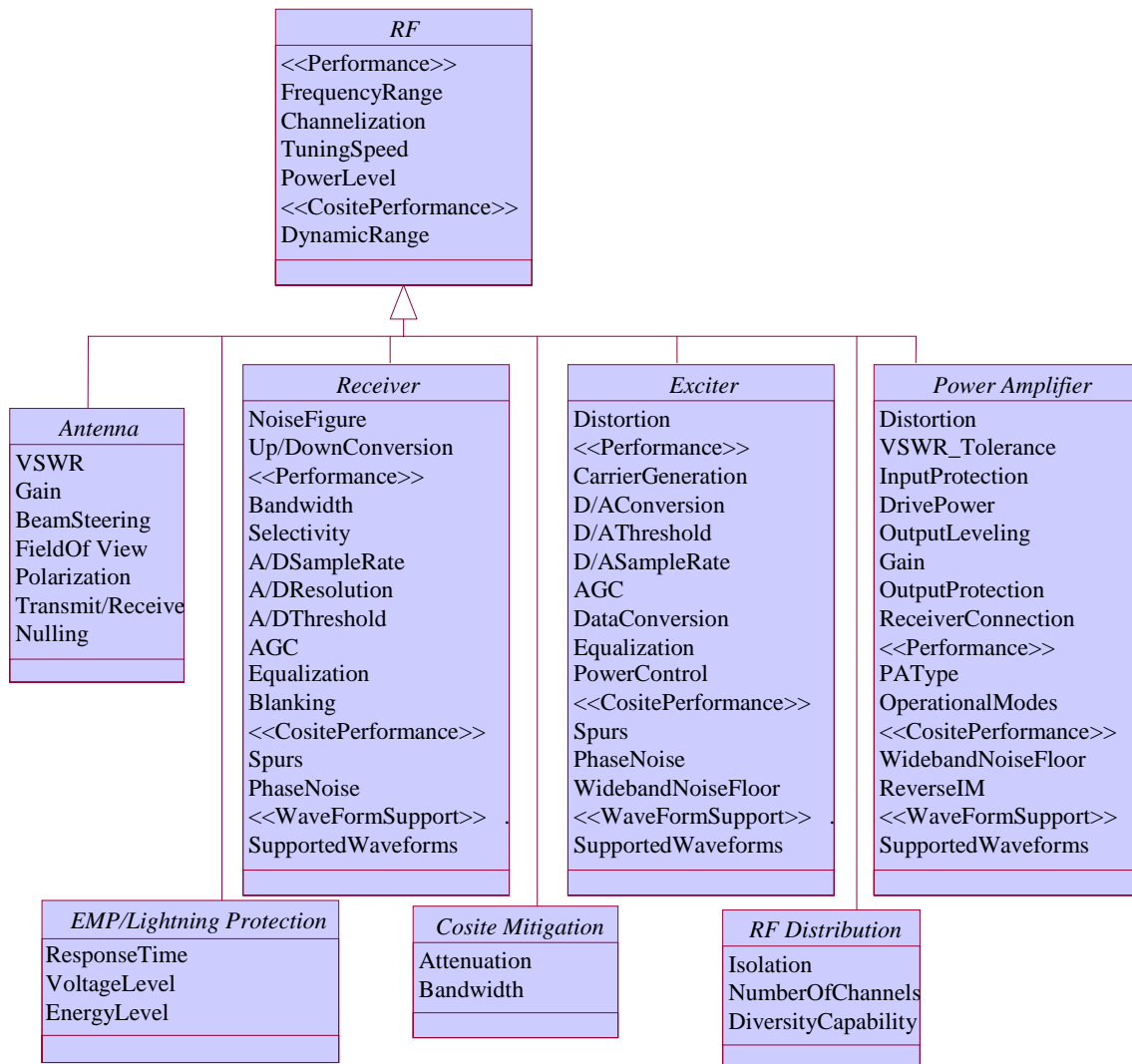
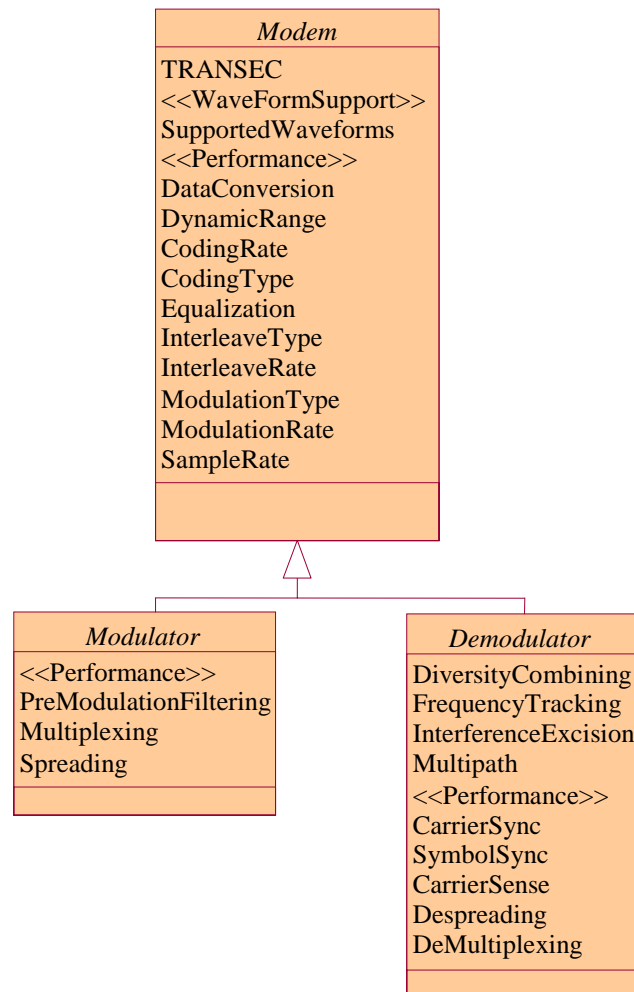


Figure 4-3. RF Class Extension

#### 4.2.3.2 Modem Class Extension.

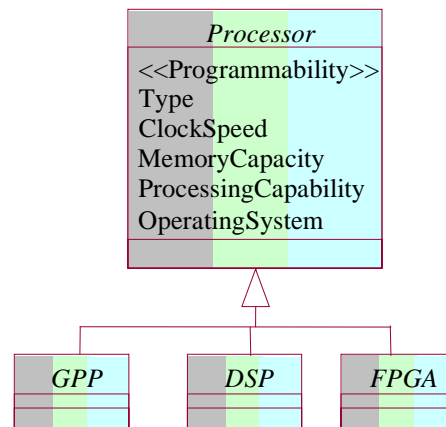
The *Modem* class shown in figure 4-4 has subclasses that encapsulate the attributes of modulation and demodulation functions. The *Modem* class also contains attributes that can be used to describe the range of signal processing and data conversion capabilities such as spreading and de-spreading. The <<WaveFormSupport>> stereotype labels the attribute of SupportedWaveforms. This is an attribute indicating specifically what waveforms the modem is capable of performing.



**Figure 4-4. Modem Class Extension**

#### 4.2.3.3 *Processor* Class Extension.

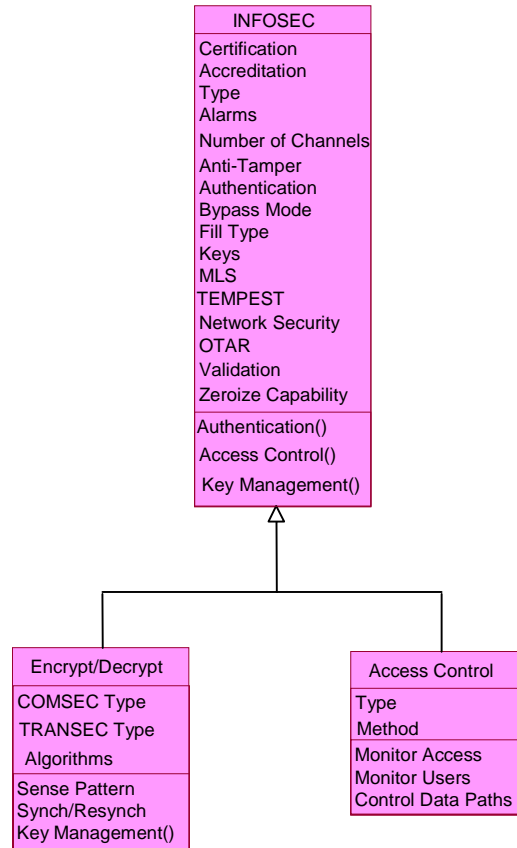
The *Processor* class shown in figure 4-5 directly supports software operations by providing the processors, memory, and supporting functions. Devices derived from this class include General Purpose Processors, Digital Signal Processors, and extend to modules utilizing programmable logic devices (FPGAs, etc.). The class captures the attributes of processing devices needed by the system resources. This *Processor* class represents the type of hardware that, in itself, essentially has no unique radio-functional capabilities of its own. Its actual use, or personality, is a function of the software that is loaded into and executed on it. It can be envisioned that as processor speeds and software capabilities evolve, this class of hardware will tend to dominate future radio systems while some of the other hardware specific functions will be replaced by processors and software. As this happens, the attributes associated with function and performance will effectively migrate to the software applications that are running on the host processors.



**Figure 4-5. Processor Class**

#### 4.2.3.4 INFOSEC Class.

The *INFOSEC* class provides structure for a hardware device that is described by the type of cryptographic features it supports and certifications for which it has been qualified. Figure 4-6 lists *INFOSEC* class attributes.

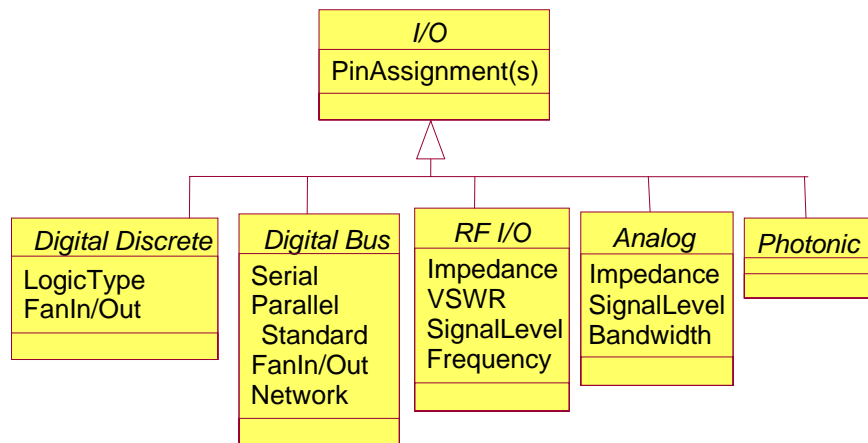


**Figure 4-6. INFOSEC Class**

#### 4.2.3.5 I/O Class Extension.

The *I/O* Class shown in figure 4-7 provides representation for general physical connectivity and is not limited to just user interfaces.

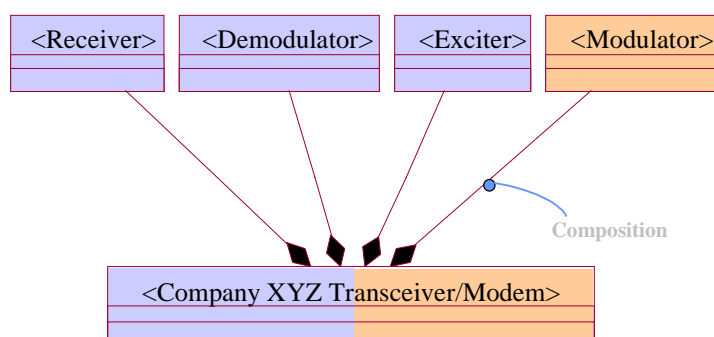
For every hardware device, the critical interfaces are those that are presented to the “outside world”. The definition of a critical interface is dependent on the class hierarchy level at which the hardware device is being viewed. For example, if the HW device is a complete radio system, it inherits attributes from the chassis class and its critical interfaces are defined at the chassis physical boundary. Additionally, each module within the radio system has critical interfaces unique to it; and its I/O attributes are inherited from the *I/O* subclass.



**Figure 4-7. I/O Class Extension**

#### 4.2.4 Attribute Composition.

As hardware technology evolves, hardware modules will encompass increased levels of functionality due to higher levels of integration. This will allow more functional hardware classes to be realized within individual physical hardware modules. The function of the individual classes remains the same, but they are physically realized on the same circuit card or module. UML provides the 'composition' relationship to represent this. An example of this is shown in figure 4-8, showing a module that provides receive, transmit, and modulation/demodulation capabilities, and using the hardware class model to illustrate this fusion of capabilities. The resultant attribute list for the module will be composed of the unique mix of features encapsulated by the four hardware classes from which it is composed. Since each of the individual classes inherit attributes from its respective higher level class, the hardware module also inherits from the higher levels.



**Figure 4-8. Typical Hardware Device Description using the SCA HW Class Structure**

### 4.3 DOMAIN CRITERIA.

As communications systems assume multi-band, multi-channel, and multi-mission capabilities, a dilemma arises. When trying to satisfy the needs of both the small, highly mobile user (Handheld Domain) and the large command center (Maritime/Fixed Domain), it is evident that distinctly different mission and platform constraints exist. Offering the same solution for both extremes is obviously not the optimum – or cost effective – solution for either. The highly-mobile user requires a compact, environmentally-robust terminal containing embedded message processing, sized sufficiently to their needs, but not so large as to meet the intensive filtering/formatting/networking needs of the command center. The command center, on the other hand, requires environmental robustness only to the inhabited level. There are many, real barriers to complete commonality - cost being the largest. The most significant hardware cost-savings potential is the use of COTS standards, technology, and components, where possible. The SCA provides the standard for use of COTS technology, design reuse across products, and an open, well-documented architecture allowing multiple contractors to implement an entire system or only a portion of it.

### 4.4 PERFORMANCE RELATED ISSUES.

A particular implementation of the SCA can have significant impact on the equipment performance, especially in the case of complex waveforms and multi-channel radios. The areas



of cosite performance and system control timing have been identified as two key performance areas for careful consideration. Discussions of the cosite effects and mitigation techniques applicable to the physical implementation of the architecture are in the SRD.

#### **4.5 GENERAL HARDWARE RULES.**

Requirements placed on hardware objects by the SCA reflect a balance between the need to support extendibility and interchangeability, and the support of technology growth and domain constraints. The result is a limited set of specific rules (listed below) augmented by implementation guidelines, much of which is in the SRD.

##### 4.5.1 Device Profile.

Each supplied hardware device shall be provided with its associated Domain Profile files as defined in section 3.1.3.4, Domain Profile. These files shall be in XML, using the format shown in Appendix D.

##### 4.5.2 Hardware Critical Interfaces.

###### 4.5.2.1 Interface Definition.

Hardware critical interfaces shall be defined in Interface Control Documents that are available to other parties without restriction. Critical interfaces are those interfaces at the physical boundary of a replaceable device that are required for the operation and maintenance of the device.

###### 4.5.2.2 Interface Standards.

Hardware critical interfaces shall be in accordance with commercial or government standards, except as allowed below.

An exception is allowed to this requirement, if there are program performance requirements that require a non-standard interface. In this case, the interface definition shall be clearly and openly documented to the extent that interfacing or replacement hardware can be developed by other parties without restriction.

###### 4.5.2.2.1 Interface Selection.

In addition to the above, interface selection should consider the availability of supporting products that have wide usage, are available from multiple vendors, and are expected to have long-term support in the industry.

##### 4.5.3 Form Factor.

The form factor of the hardware objects should be, where practical, in accordance with commercial standards.

##### 4.5.4 Modularity.

The partitioning of the hardware architecture into modules should be chosen to allow for ease of upgrade through technology insertion or replacement of modules based on form, fit, and function. Module boundaries are critical interfaces as defined in 4.5.2.1.



## **5 SECURITY ARCHITECTURE DEFINITION**

*{ This section will contain access control and authentication requirements applicable to the general SCA. JTRS security definitions and requirements are contained in the Security Supplement to the SCA Specification. }*



## **6 COMMON SERVICES AND DEPLOYMENT CONSIDERATIONS**

### **6.1 COMMON SYSTEM SERVICES.**

This section will define any common system services that are not part of the CF but are considered part of the SCA. None have been identified at this time.

### **6.2 OPERATIONAL AND DEPLOYMENT CONSIDERATIONS.**

This section will address common interfaces or features necessary to support deployment of SCA-compliant systems in the field. None have been identified at this time.



## **7 ARCHITECTURE COMPLIANCE**

This section defines the criteria for certifying candidate system, hardware, and software application products to this specification.

This specification may be applied to procurement of a multitude of radio products and communication systems. In addition, this specification may also be applied to hardware-only or software-only products that would be hosted on SCA-compliant systems.

### **7.1 CERTIFICATION AUTHORITY.**

The JTRS Joint Program Office (JPO) holds the authority to certify that a candidate product meets the requirements of this specification. This authority may be transferred, in time, to a general standards body.

### **7.2 RESPONSIBILITY FOR COMPLIANCE EVALUATION.**

The responsibility for performing the evaluation of a candidate product's compliance is TBD. This body will determine the test methods and procedures used to establish compliance.

### **7.3 EVALUATING COMPLIANCE.**

Compliance to this specification is defined as meeting all requirements, except as specifically allowed herein. Products submitted as "SCA-Compliant" will be evaluated for compliance in accordance with the test methods and procedures established per section 7.2.

### **7.4 REGISTRATION.**

Documentation of some elements of an SCA implementation, as defined in sections 3 and 4, will be submitted to a Registration Body to be established, initially, by the JTRS JPO.

[The establishment, membership, rules, and operation of Registration Bodies are beyond the scope of the SCA.]

Some elements of an SCA implementation are identified with a UUID. As used in this specification, the UUID is defined by the DCE UUID standard (adopted by CORBA). (OSF Distributed Computing Environment, DCE 1.1 Remote Procedure Call) No centralized authority is required to administer UUIDs (beyond the one that allocates IEEE 802.1 node identifiers [Medium Access Control (MAC) addresses]).

